View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Radboud Repository

Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a publisher's version.

For additional information about this publication click this link.
https://repository.ubn.ru.nl/handle/2066/233620

Please be advised that this information was generated on 2021-11-04 and may be subject to change.

THE HAPPY, THE SAD, AND THE UNKNOWN

Program Analysis and Automated Scheduling
For Fault-Tolerant Workflows

Proefschrift
ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M. van Krieken,
volgens besluit van het college van decanen
in het openbaar te verdedigen

op donderdag 10 juni 2021
om 12:30 uur precies

door
Markus Alexander Anton Klinik
geboren op 16 april 1981 te Nürnberg, Duitsland

Promotor

Prof. dr. ir. M.J. Plasmeijer

Copromotor

Dr. J.M. Jansen	(Nederlandse Defensie Academie)

Manuscriptcommissie

Prof. dr. S.B. Scholz	
Prof. dr. F. Henglein	(Københavns Universitet, Denemarken)
Prof. dr. J.T. Jeuring	(Universiteit Utrecht)

This research has been funded by the Netherlands Defence Academy and TNO.

ISBN: 978-94-6419-200-1

Abstract

The Manning and Automation project of the Royal Netherlands Navy is an effort to modernize the mode of operation on board of their ships. The central idea is that with more automation and highly trained crew members, the same level of service can be provided with a smaller crew. A well-known saying in Navy circles is: "No plan survives the first contact with the enemy". One part of the Manning and Automation project, and this is where my research question originated from, is to make plans and standard procedures more flexible, more fault-tolerant, and more resilient against external disturbances. Plans are always executed to achieve a goal. Making a plan more resilient means that its goal will still be reached, even if the original plan can no longer be executed as intended. There are many reasons of different nature why plans become invalid, and each nature needs to be addressed with a different approach. I focus on a specific kind of plan, namely workflows. In this thesis I investigate which techniques from computer science, and more specifically programming language research, can be leveraged and applied to workflows. Correctness and robustness of computer programs is a field of active research in the programming language community, and if we regard workflows as a kind of program, many existing techniques can be transferred from programs to workflows.

The techniques I adapt from programming language research are: task-oriented programming, correctness with regard to a specification, formal programming language semantics, static program analysis, symbolic execution, genetic algorithms, and automated scheduling. This results in different methods, shining light from different angles on the central purpose of making plans reach their goal. I have implemented a static analysis that determines, given a workflow in the form of a task-oriented program, how much resources the workflow needs in the worst case. An extension of this analysis additionally calculates the points in time where these requirements occur. This technique is then applied in an energy consumption analysis of programs in a C-like imperative programming language. The question of whether a plan reaches its goal in the first place, is tackled by the design and implementation of symbolic execution engine for TopHat, a minimal semantics and implementation of task-oriented programming. Finally, I implemented a scheduler that, given a partially ordered set of tasks to execute, their resource requirements, available resources, and arbitrarily complex constraints on all of those, computes an assignment of resources to tasks and start times for all tasks.

All of this is made possible by task-oriented programming. Task-oriented programming allows workflows to be represented as computer programs, which in turn allows me to address the question of whether a plan reaches a goal using well-known programming language techniques. I have done so in three different ways, with static analysis, symbolic execution, and automated planning.

Samenvatting

Het Manning and Automation project bij de Koninklijke Marine heeft als doel de werkwijze aan boord van marineschepen te moderniseren. Het kernidee is dat de marine door automatisering en hoogopgeleide mariniers hetzelfde niveau van dienstverlening kan bieden. Een bekend spreekwoord bij de marine is ”Geen plan overleefd het eerste contact met de vijand”. Daarom is een onderdeel van het Manning and Automation project, en het deel waar mijn onderzoeksvraag vandaan komt, hoe je plannen en standaardprocedures flexibeler, fouttoleranter, en robuuster kunt maken tegenover externe verstoringen. Plannen worden altijd uitgevoerd om een doel te bereiken. Een plan robuuster maken betekent dat men dat doel nog steeds kan bereiken, zelfs als men het plan niet kan uitvoeren zoals oorspronkelijk bedoeld. Er zijn verschillende manieren waardoor een plan ongeldig kan worden. Elke manier moet op eigen wijze worden aangepakt. Ik beschouw een specifieke categorie van plannen, namelijk workflows. In dit proefschrift onderzoek ik welke technieken uit de informatica, en specifieker uit het onderzoek van programmeertalen, men op workflows kan toepassen. Correctheid van programma's is een veld van geavanceerd onderzoek in de programmeertaalwetenschap. Wanneer we workflows als een soort programma beschouwen, kunnen we veel bestaande technieken van programma's op workflows toepassen.

De technieken die ik uit de wereld der programmeertalen toepas omvatten: taakgeörrienteerd programmeren, correctheid met betrekking tot een specificatie, formele semantiek van programmeertalen, statische programma analyse, symbolische executie van programma's, genetische algoritmen, en geautomatiseerde roostering. Zo kijk ik vanuit verschillende hoeken naar de centrale vraag hoe je plannen hun doel kunt laten bereiken. Ik heb een statische analyse geïmplementeerd die, gegeven een workflow in vorm van een taak-georiënteerd programma, bepaald hoeveel middelen hij in het slechtste geval nodig heeft. Een uitbreiding van deze analyse berekent bovendien op welke tijdstippen de middelen tijdens de uitvoering van de workflow nodig zijn. Deze techniek pas ik vervolgens toe op een iets andere domein: namelijk het energieverbruik van imperatieve programma's. De vraag of een programma zijn doel überhaupt kan bereiken, wordt nagegaan door het ontwerp en de implementatie van een symbolische executie van TopHat, een minimale semantiek van taak-geörrienteerd programmeren. Ten slotte heb ik een algoritme geïmplementeerd dat een indeling van middelen en een roostering van alle taken berekent. Dit gegeven een partieel geordende verzameling van taken, de middelen die ze vereisen, beschikbare middelen, en arbitraire beperkingen daarop.

Dit alles wordt mogelijk gemaakt door taak-geörrienteerd programmeren. In taakgeörrienteerd programmeren worden workflows als programma's gerepresenteerd, wat toestaat dat ik de vraag of een plan zijn doel bereikt met algemeen bekende programmeertaaltechnieken kan aanpakken. Ik heb dit op drie verschillende manieren gedaan, met statische analyse, symbolische executie, en geautomatiseerd roosteren.

Contents

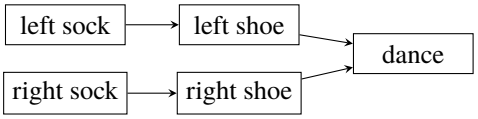
1 Introduction

1.1 Reaching the Goal

When executing a plan in order to reach a goal, three things can happen. First either everything goes according to plan, second an anticipated error occurs, or third an unanticipated error occurs. Let us call these the *happy case*, the *sad case*, and the *unknown case*. For each case we face the question: will the plan reach the goal? Before we delve further into this question, we have to look at what exactly plans are.

In general, executing plans covers any situation where stepwise actions are being carried out. In this thesis, *plans* are computer programs and workflows. Computer programs are plans in this sense because they codify computation steps with the goal of calculating a result. Workflows are plans because they specify tasks to be executed and their ordering, to achieve some desired effect. Workflows and programs can be combined by allowing workflows to be specified in an algorithmic manner, in the form of computer programs that contain statements that represent tasks. This style of programming is called task-oriented programming, and this thesis is written with a focus on it. We use the terms *plan*, *program*, and *workflow* interchangeably, as they are the same in task-oriented programming. Figure 1.1 shows a workflow, specified with boxes and arrows, and the equivalent task-oriented program. Whenever we talk about workflows, plans, or programs, readers are invited to imagine whichever representation they are more comfortable with.

No matter how plans are represented, executing them requires *resources*. When hearing the word "resource", most computer scientists might think of what they learned in their course on operating systems. There, resources were things inside a computer, like file descriptors, memory, input- and output devices, or the CPU itself, and the main concern was that concurrent processes should not interfere with each other when sharing resources. We use the word differently in this thesis. We use it to refer to real-world resources like people, tools, power, or fuel, whatever is required to carry out work. For example, extinguishing a fire requires a fire fighter and a fire extinguisher. Our main concern is whether resources are available in large enough quantities for a given plan to be executed. Plans often have




```
(  (leftSock >> leftShoe)
&& (rightSock >> rightShoe)
) >> dance
```

Figure 1.1: Different forms of the same plan: A workflow and its corresponding task-oriented program

multiple ways of executing them, different paths from start to finish that are followed under certain conditions. Some parts of a plan deal only with the happy case, others handle unhappy cases. For each case the question of whether the goal can be reached takes a slightly different form.

The part of a plan that handles the happy case is called the *happy flow*. The happy flow assumes that all its actions will complete successfully, and that no external disturbances occur. It does not include error handling or -recovery. Even in the happy flow it may not be obvious that the goal will always be reached, because plans can get complicated when they represent real-world scenarios. For the happy case, the question of whether the goal will be reached translates to: Is the happy flow correct?

The parts of a plan that handle sad cases are called the *sad flow*. It is possible to plan ahead and work around anticipated external disturbances. For example, plans can contain error recovery procedures, or switch to alternative plans consisting of different, possibly more expensive actions. In addition to correctness, alternative plans should not exceed the available resources, especially when the alternative versions of multiple concurrently running plans spring into action. For the sad case, the question of whether the goal will be reached translates to: Are there enough resources available to handle the sad flow?

There is no flow for the unknown case. External disturbances that are unknown at planning time need to be addressed differently. No amount of pre-planning can prepare for them, because then by definition they would fall into the category of anticipated disturbances. Unknown disturbances can only be addressed when they occur during plan execution, when their nature becomes clear. Reaching the goal when facing an unknown disturbance requires flexibility of the plan, re-planning and re-scheduling, out-of-the-box thinking, improvisation, and creative problem solving. The question of whether the goal will be reached in the unknown case translates to: Can the plan be repaired?

This thesis consists of three parts, each examining one of the questions. In each part we take a known topic from the computer science literature and show how it can be used to answer the question. The question of whether a given plan is correct is studied in the part about formal program correctness. The question of whether enough resources are available for a given plan is studied in the part about static program analysis. Finally, the question of whether a plan can be repaired is studied in the part about dynamic resource management. The following table summarizes this organization.

Case	**Question**	**Thesis topic**
Happy	Is the plan correct?	Formal correctness
Sad	Do we have enough resources?	Resource analysis
Unknown	Can the plan be repaired?	Dynamic resource management

1.2 The Manning and Automation Project

The research underlying this thesis was funded by the Royal Netherlands Navy, within the scope of their Manning and Automation project. Figure 1.2 shows one of the ships of the Royal Netherlands Navy that was sometimes talked about in the project's meetings. The Manning and Automation project has two main goals. The first goal is to bring current state-of-the-art communication, information, and optimization techniques on board of navy

Figure 1.2: The HNLMS Groningen, a Holland-class offshore patrol vessel

ships. Navy ships have a life span of about 40 years, which means that the ones currently in use employ decades old technology. Upgrades can only be done on a superficial level, for example by installing new computers, but the core structure of battle management stays antiquated, with speaker broadcasts and paper-based communication, ill-suited for the claustrophobic metal hallways and rooms on a ship. The Manning and Automation project aims to redesign mission management from the ground up. This includes, for example, modernizing the interior design of the command bridge, installing IT infrastructure and integrating all systems and processes with it, and making workflows of standard procedures more flexible.

This last point correlates with the second goal of the Manning and Automation project, which is to increase automation to allow for reduced manning. This is where my work fits into the picture. Where currently many tasks have dedicated crew members, the project aims to employ less, but highly trained personnel, that can switch roles on a moment's notice, as required by the priorities of a given situation. With support of an integrated mission management system, which is aware of the location of crew members and the status of equipment, it should be possible to derive decision support for the commanding officer. The research topics of this thesis, resource analysis, resource management, and workflow correctness, all relate to this second goal in one way or another.

1.3 Background

We use a variety of techniques from the field of computer science in our venture to answer the research question. In this section we briefly look at each of them, and discuss their relevance for our purpose.

1.3.1 Task-Oriented Programming

Of the seven publications that this thesis is based on, four are about task-oriented programming (TOP). TOP is a programming paradigm for implementing multi-user collaboration applications. The goal of TOP is to provide building blocks that are always needed for creating such applications, so that they can be rapidly implemented and easily maintained. The central concept of TOP is that of *tasks*. Tasks in a TOP program represent work to be done in the real world. There are two kinds of tasks, basic tasks and compound tasks. Basic tasks represent indivisible single steps that workers can complete without further explanation, while compound tasks consist of sub-tasks, to be executed in a certain order. For example, the following TOP program asks users to enter their name and age in parallel (`-||-`), which allows them to switch between these tasks as they see fit.

```
enterInformation "Name" -||- enterInformation "Age"
```

Note that this program does not specify in any way how the tasks are represented to users, nor how exactly users will be able to switch between them. This is what makes the specification declarative. Declarative means that programmers specify *what* needs to be done, but not *how*. An execution engine takes such declarative descriptions and turns them into running applications, thereby taking care of the *how*. The biggest, most feature-complete execution engine for TOP is iTasks, described later in this chapter. There are other TOP execution engines, namely mTasks for embedded systems [Koopman et al., 2018], and our own TopHat, for the study of formal properties of TOP programs chapters 5 and 6. TOP, as a general concept independent of any execution engine, stands on four pillars, all supporting the goal of implementing multi-user applications.

The first pillar is a declarative workflow specification language. The most important elements of this language are editors and combinators. Editors represent interaction points between the workflow and the end user, and allow users to view, enter, or update data. On the user's screen they are displayed as input fields, check boxes, or any other widget suitable for editing a piece of information. Combinators allow construction of larger tasks from smaller ones, for example through parallel or sequential composition. Tasks constructed in this way can again be combined with other tasks through combinators, thereby forming whole applications.

The second pillar are shared data sources (SDS). Multi-user collaboration requires that the right people have access to the right information at the right time. SDSs allow any part of a compound task to refer to the same piece of information, which makes it possible to exchange data between people working on different subtasks. For example, one person can compose a piece of text, and after that another person can review it. It is even possible for multiple people to work with the same shared data source at the same time, for example two people can work together to fill in a damage report form.

The third pillar is functional programming. All of the above is embedded in a functional programming language, so that the benefits of functional programming are available to task-oriented programmers. This includes for example the robustness guaranteed by strong typing, and the modularity permitted by first-class and higher-order functions.

The fourth pillar is automatic creation of user interfaces, which the execution engine derives from the task specifications. The engine that runs task-oriented programs turns basic tasks into graphical control elements, with which users can interact. For example, an

editor for a number gets turned into an input field that only allows numbers to be entered. A data structure representing a damage report gets turned into a complete input screen for damage reports. Task combinators determine how the control elements are laid out on the screen. When two tasks are composed in parallel, their control elements are displayed next to each other. When two tasks are composed in sequence, they are displayed in sequence, separated by a next-button. All of this happens automatically, with programmable tweaks if desired. This lets programmers focus on the task structure itself, and not worry about the user interface, which is otherwise a major concern in traditional GUI programming. Changes in the task structure are reflected in the user interface without additional effort.

The ideas that underlie, and ultimately culminated in TOP have been in development since the early 2000's. Generic programming was added to Clean around 2001 [Alimarine and Plasmeijer, 2001]. Generic, type-based generation of graphical editors for arbitrary data types was conceived in 2003 with the GEC toolkit [Achten et al., 2003], at that time for native desktop applications. The term TOP had not been coined at that time. A combinator library for GEC, based on Arrows, was first presented in 2004 [Achten et al., 2004]. It allows composition of editor components into larger blocks, with appropriate threading of values between them. These techniques were applied to web applications in 2005 with the iData Toolkit [Plasmeijer et al., 2005; Plasmeijer and Achten, 2005]. The iData toolkit does not support control flow like stepping between screens of an application with a next button. A small challenge by Phil Wadler to include such stepping initiated the development of a monadic combinator library on top of iData, which resulted in the first version of iTasks [Plasmeijer et al., 2007]. The second version of iTasks was published in 2009 [Lijnse and Plasmeijer, 2009]. This version makes two contributions towards the development of what would later be called TOP. First, it introduces the current architecture of iTasks, which consists of a server providing a JSON web-API, and a JavaScript client accessing it. Updates of the user interface as workflows make progress can now happen incrementally, whole-page reloads are no longer needed. Second, this version focuses on declarative task specifications, where all implementation details of the iTasks execution engine are hidden from the programmer. The programmer using iTasks does not see any references to HTML or JSON when constructing tasks, this all happens behind the scenes. Finally, TOP itself was introduced in 2012 [Plasmeijer et al., 2012]. This paper presents the third iteration of iTasks, introduces shared data sources for inter-task communication, and emphasizes a user-centric view on task specifications. The last point means that task specifications express what the end user has to do, as opposed to what the program does. While already partially present in previous versions of iTasks, the third version consistently renames all tasks, so that instead of having program-centric names like "showMessage", tasks have user-centric names like "viewInformation" and "enterInformation". This paper also establishes TOP as a programming paradigm, explaining the tools and mindset a task-oriented programmer should employ, and positions iTasks as an implementation of TOP.

Most of the papers that form the basis of this thesis relate to TOP. Everybody in our research group who has worked on or with iTasks has their own idea about what TOP really *is*. We noticed this when one of my co-authors posed a small modelling challenge to our colleagues. The task was to model a simple workflow about handling customer complaints in a company, in an idiomatic task-oriented manner. We were surprised by how different the solutions were, which sparked the investigation of the *essence* of TOP. This resulted in the two publications about TopHat. The papers about resource analysis came from the question

of how military navy operations could benefit from TOP. The papers about scheduling were a result of me participating in the ongoing effort of the Netherlands Defence Academy to develop an integrated mission management system using the principles of TOP.

1.3.2 Formal Program Correctness

We are interested in the correctness of workflows. Workflows and programs are the same in task-oriented programming, which allows us to brazenly help ourselves from the buffet of program correctness. Results from this field can be applied, with appropriate modifications, to workflows. If we understand how to prove programs correct, we can prove workflows correct.

What does it mean for a program to be correct? One possible answer is: A program is correct when it does what its programmer wants. Unfortunately, computers can not read minds, at least not with currently available technology. This gap between the programmer's mind and the program prevents us from accepting this definition, we have to narrow it down. Until the arrival of mind-reading technology, we must require programmers to explicitly state their intention in some form of specification. They have to *say* what they want. This allows us to at least verify that the program does what programmers *says*, but there is still a gap between what they *say* and what they *want*. This is like the wife who tells her husband to not spend so much time in the office. When he comes home the next day, he tells her that she will be pleased to hear that he became a member of the local golf club. Even if a program fulfils its specification, nobody can guarantee that the specification itself does not have bugs. Unless we want to descend into the rabbit-hole of specifications for specifications, we must trust the ability of programmers to produce specifications that faithfully reflect their intentions. This leads us to the commonly accepted definition of program correctness: A program is correct if it fulfils its specification. Drilling into this definition raises further questions: What is a program? What is a specification? How to prove adherence to a specification?

What is a program, then? If we don't know what a program is, we can identify it with something of which we know what it is. One way to do this is to identify a program with the effect it has on a computer's internal state when compiled to machine code and executed. Specifying the behaviour of a program in this sense requires programmers to talk about details of the hardware architecture, about bits, registers, and memory locations. For some programs, this is the way to go, for example when verifying that device drivers do not corrupt the kernel's memory. For high-level specifications however, this defeats the purpose of having high-level languages in the first place, which is to abstract away from such details, and relieve programmers from the burden of thinking about hardware. Programmers think in abstract terms when programming in high-level languages. Their actual requirements are concerned with algorithms and data structures, not CPU instructions and memory locations.

For this reason it is better to identify a program with an abstract mathematical model that captures all relevant aspects but leaves out implementation details. Instead of translating programs to hardware, of which we have a concrete understanding, we translate them to mathematical constructs, of which we have a clear understanding. Different mathematical models may be appropriate for different programming languages. Models found in the literature include finite state machines, labelled transition systems, petri nets, process algebras, structural operational semantics, denotational semantics, or axiomatic semantics.

Once a program is translated in this way, we have a mathematical object in our hands which we can study and of which we can determine if it has certain properties. But how do we specify such properties?

Specifications can be given in a logic that talks about the mathematical model. The logic used to formulate specifications depends on the choice of the mathematical model. Popular logics include temporal logic, propositional logic, first-order logic, or variants thereof. Such a logic allows precise specification of desired properties, for example that something good eventually happens, or that something bad never happens. Finally, we need a mathematically sound method to prove that the mathematical model adheres to the mathematical specification. There are three classes of methods found in the literature.

With *deductive methods*, proof obligations are derived from the program and the specification, which are then discharged using an inference system like natural deduction. The proofs can be carried out with pen and paper, or with the help of interactive theorem provers. An example for a deductive method is Hoare-style axiomatic semantics. It can be used to derive invariants and post-conditions A from a program. An inference system can then be used to discharge the proof obligation $A \to S$, namely that the specification S is a logical consequence of the invariants. An illustrative example can be found in [Nielson and Nielson, 1992], where an axiomatic semantics and a pen-and-paper proof is used to prove that a given algorithm computes the factorial function. Using this method is difficult, because it cannot be completely automated. The programmer must manually come up with loop invariants and post-conditions at certain points in the proof, which requires insight and ingenuity.

Model checking uses exhaustive state space exploration to prove by inspection that every state a program can be in fulfils the specification. Model checking can be employed when finite state machines or labelled transition systems are suitable models for the programming language under consideration. For example, students of the Operating Systems course at Radboud University are asked to model the synchronization algorithms by Downey [2008], which are given in Python, as finite state machines in UPPAAL [Larsen et al., 1997], then formulate the synchronization requirements in UPPAAL's version of temporal logic, and let UPPAAL show that the model fulfils the requirements. Model checking is completely automated. Once the programmer creates the model and the specification, the proof runs unsupervised. This comes at the cost of memory and time, because the state space to be explored can grow quite large.

Static analysis comes in the form of alternative program semantics that execute the program in an abstract manner, keeping track of only properties of interest. These semantics should be coarser than the actual semantics to be computable and always terminating, but fine grained enough to imply the desired property. Static analysis is different from the above described methods in that programmers cannot formulate their own specifications. Static analysis calculates pre-defined properties. For example, for imperative programming languages, *live variable analysis* determines for each program point, which variables will be used on some path after that point. For higher-order functional or object-oriented programming languages, *control flow analysis* determines for each function call which function might actually be called [Nielson et al., 1999]. The Astrée analyser [Cousot et al., 2005] implements several static analyses that can prove absence of runtime errors in C programs, like integer over- and underflows.

To summarize, formal correctness needs a translation from the actual requirements to a

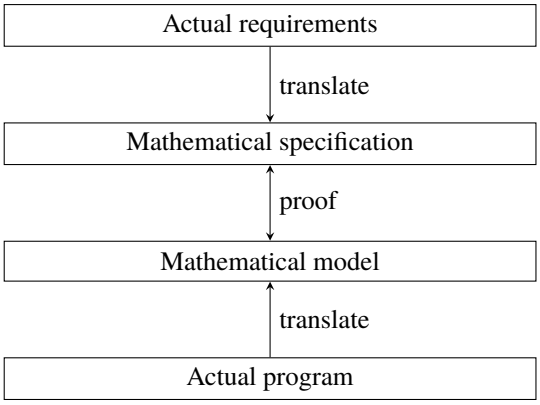


Figure 1.3: Schematic overview of formal correctness proofs

mathematical specification, from the actual program to a mathematical model, and a proof that the model satisfies the specification. The introduction to formal verification by Harrison [2011] illustrates the situation with a diagram similar to fig. 1.3 (some arrows reversed here). Cousot and Cousot [2010] give another nice introduction to formal verification.

1.3.3 Formal Programming Language Semantics

Formal programming language semantics are mathematical models of program execution. They are mathematical models in the sense of section 1.3.2, with the intention of describing the full computational meaning of programs. We design several small programming languages in this thesis, and we strongly believe that no definition of a programming language is complete without a formal semantics. Some people believe that the sole purpose of a programming language is to let programmers write programs, and for that language manuals are sufficient. However, programming languages can become quite complicated and acquire lots of features over their lifetime. Defining the meaning of a programming language in a natural-language document inevitably leads to ambiguities and unspecified behaviour. Maintaining a formal semantics that grows as new features enter the language forces the language designer to think critically and accurately about how features are going to fit together before implementing them. Imposing formal semantics on programming languages after the fact is a big but useful endeavour, as can be seen for example by the efforts of Park et al. [2015] and Krebbers [2015], which formalize the English-language standard documents of JavaScript and C, respectively. Both works reveal inconsistencies in the language standards, and bugs in various implementations. This demonstrates how useful formal semantics are. But what exactly are formal semantics?

Mosses [2006] breaks the term *formal programming language semantics* down as follows. *Programs* of a language are defined by their syntax. The *semantics* of a programming language defines its computational effects. This means that it defines *what* a program computes, but not *how*: it defines the relation between input and output, abstracting away from implementation and hardware details. A semantics is *formal* if it is given in a notation that already has a precise meaning. The notations for semantics commonly found in the

literature fall into three main categories.

Axiomatic semantics are used for imperative programming languages, and specify preconditions, postconditions, and invariants of the statements of the language with regard to the program variables. Axiomatic semantics was developed by Hoare in the 1960s with the goal of proving individual programs correct [Hoare, 1969].

Denotational semantics define the meaning of programs by translating programs to mathematical constructs, more specifically, elements of an algebra. It was developed by Scott and Strachey in the 1960s with the goal of providing a mathematical framework for the study of programming languages [Scott and Strachey, 1971]. Over the years they proposed several mathematical models, among them lattices, topologies, and most famously, domains. As Scott himself remarks, using denotational semantics requires a substantial amount of mathematical background, which most computer scientists do not have [Scott, 1982].

Operational semantics define the computational effects of program constructs in the form of relations between terms of the language itself. No translation to a different formalism takes place. If the relation is defined by syntax-directed rewrite rules, it is called *structural operational semantics*. There are two variants of structural operational semantics, big-step and small-step. In big-step semantics, terms are directly related to the final value they evaluate to. Intermediate steps are not visible in the relation. In small-step operational semantics, terms are related to their direct successors of taking one small evaluation step. All intermediate steps are included in the relation. Small-step operational semantics were introduced by Gordon Plotkin in an effort to treat the untyped lambda-calculus as a programming language [Plotkin, 1975], and have since then become a popular, if not the most popular, method of specifying programming language semantics [Plotkin, 2004]. Big-step operational semantics were developed by Gilles Kahn and colleagues [Despeyroux, 1984] as a variant of small-step semantics, originally for their Typol language, with the goal of specifying static verification systems.

Big-step and small-step are useful in different situations. Nielson and Nielson write that for simple languages, big-step and small-step are equivalent, but more advanced features are easier to express in one or the other [Nielson and Nielson, 1992]. Big-step cannot distinguish between looping and abnormal termination, while small-step can. For non-deterministic languages, big-step will suppress looping, i.e. it will favour terminating executions, while small-step does not. Interleaving execution of concurrent threads cannot be expressed in big-step at all, while in small-step this is easy. Local scope is easy to express in big-step, while small-step needs complicated auxiliary constructions for it. Robert Harper said, in a lecture at the Oregon Programming Language Summer School 2017 [OPLSS, 2017], that big-step is good for user manuals, but small-step is good for theory and proofs. He goes on to say that the very nature of computation is that "I'm here, I know how to take a step". In this sense, the small-step transition system is fundamental, the big-step evaluation relation is secondary.

For our various programming languages, we mostly use small-step semantics. In chapters 2 to 4, we use small-step semantics to define the dynamics of the presented programming languages. These sections do not include proofs, but the given semantics serve as an expression of intent, and as basis for the implementations. In chapters 5 and 6, we use big-step semantics for the host language, and small-step semantics for the task language. We use big-step for the host language, because the host language is terminating, so small-step and big-step are equivalent, and big-step is very close to how one would

implement an expression evaluator. We use small-step for the task language, because it deals with interleaving execution of concurrent expressions and with user input, both of which are easier to express in small-step. The symbolic execution semantics in chapters 4 and 6 are given as non-deterministic small-step semantics. Non-deterministic means that one term can have multiple successor terms.

1.3.4 Static Program Analysis

The goal of *static program analysis* is to determine properties about programs without executing them. Static analysis works solely on the syntactic structure of programs, and tries to predict some property about their execution. Static analysis is often used in compilers to determine if it is safe, or beneficial, to enable certain optimizations. For example, a variable that is read often might be kept in a register instead of memory, to speed up access to it. Another example is that reordering assignment statements can improve execution speed when it exploits caching, but is only safe to do when the second assignment does not depend on the first one.

An important class of static analyses are type checkers. For languages with type systems, type checkers verify that functions and expressions always evaluate to a value of a certain type. The type systems of languages like Java or Haskell are designed to be simple enough for their type checkers to be automatic and fast. However, there are languages whose type systems are so expressive that type checking becomes undecidable. In these languages the programmer has to manually provide hints, or even full proofs, for the type checker. Nonetheless, these type systems still count as static analyses, because they determine a property of the program without executing it.

Static analysis always computes an overapproximation of the property in question. Due to the limits of computability, i.e. the undecidability of the halting problem, ultimate confidence of whether a program actually has a desired property can only be obtained by running the program with every possible input. This is impractical, as some runs might take a long time to complete, or might not even terminate at all. The compromise that all static analyses make is to trade accuracy for guaranteed termination. This means that analysis results will always overapproximate the real behaviour, erring on the safe side. For example, type checkers sometimes reject type-correct programs, but they never accept incorrect ones. Consider the Haskell expression `if True then "hello" else 0`. This expression always evaluates to `"hello"`, so its type could safely be String, but Haskell's type checker will reject this expression. In this case, the condition is constant `True`, but in general it is not possible to decide whether the condition will always evaluate to the same value. This is why type checkers demand both branches of conditionals to be of the same type, so no matter what, the whole expression evaluates to a value of the same type.

This example is an instance of a general principle of static analysis. The analysis result of conditionals is the *merge* of the results of the branches. Different analyses use different merge operations. For type checkers, merging is a test for equality. For analyses that yield sets of variables this could be the intersection or union, depending on the analysis. Merging of paths occurs in other places as well, whenever multiple branches of control flow converge. This happens for example at the end of while loops, or at function call sites. Merging of paths leads to overapproximation, and can result in combined predicted behaviour that neither branch on its own exhibits. For example, if one branch has live variable x, and

another branch has live variable y, then their merge is the set $\{x, y\}$.

Many static analyses for imperative programming languages follow a similar pattern, differing only in certain details. The control flow of imperative programs is explicit, so static analysis can be conveniently based on their control flow graph. Nielson et al. [1999] describe a generic technique called *monotone frameworks*, where programs are decomposed into basic blocks, with connections between them. Each block has an input and an output node, and inputs and outputs of blocks are connected. Sometimes outputs of multiple blocks are connected to the same input of another block, or the output of one block to inputs of multiple other blocks. The tentative analysis result is propagated along connections, and each node can transform it. Some analyses need this to happen in a forward manner, others in a backward manner. When the outputs of multiple blocks are connected to one input of another block, merging is performed. Monotone frameworks can be parametrized with different transformation and merge operations, called *transfer functions* to compute different analyses. For example, an assignment node can add the assigned variable to a set, and merge can take the union of multiple sets. The parametrized monotone framework gives rise to a set of constraints that describe the output of each block in terms of its inputs, and the connection between outputs and inputs of different blocks. The final analysis result can be computed by repeatedly applying the effects of the constraints, updating the value at the corresponding block's output node. If the transformation and merge operation fulfil certain monotonicity properties, this process is guaranteed to terminate.

Functional programs do not have explicit control flow like imperative programs do, so monotone frameworks are not applicable. Instead, it is much more natural to piggy-back static analyses on top of the language's type system. For example, in a functional language with exceptions, an exception analysis can be implemented on top of the type system, so that, in addition to determining the type of an expression, it can determine the set of exceptions the expression possibly throws. This additional information is called the *effect* of the expression. Merge operations are needed for combining the effects of subexpressions of an expression. For example, a conditional whose then-branch has effect $\{X\}$, i.e. it can throw exception X, and else-branch has effect $\{Y\}$, has combined effect $\{X, Y\}$, which means the whole conditional possibly throws either.

In chapters 2 and 3 we use techniques from monotone frameworks and effect systems to predict the resource consumption of task-oriented programs. Task-oriented programs are a combination of imperative and functional programming, where the task layer, an imperatively-flavoured monadic sublanguage, is embedded in a functional host language. Our effect system searches for task expressions in programs and collects sets of constraints that encode resource consumption in formulas similar to the ones obtained from monotone frameworks. These constraints are stored in the types of expressions and function definitions. The constraint set that is the result of analysing the main function is then solved using the constraint solver of monotone frameworks. This is possible because our resource constraints fulfil the required monotonicity properties.

1.3.5 Symbolic Program Execution

Symbolic execution is a technique for bug finding and software verification that sits between testing and static analysis. Testing *executes one possible path* of a program with concrete inputs. This makes it difficult to achieve high code coverage, but never gives false positives.

Static analysis *examines all possible paths* without any inputs. It achieves perfect coverage, but suffers from false positives caused by overapproximation. Symbolic execution *executes all possible paths* of a program by supplying it with *symbolic inputs*. It achieves high code coverage and high precision, which means no false positives, at the cost of high complexity. The goal of symbolic execution is to find failing programmer-defined assertions, or predefined programming errors like index out-of-bounds errors or division by zero.

Originally invented in the 70s [King, 1975, 1976], symbolic execution has received increased attention in recent years [Baldoni et al., 2018] due to advances in computer technology and SMT solving. Increased memory capacity and decreased memory cost make it viable to explore large state spaces with off-the-shelf computers. SMT solvers that exist today are powerful enough to solve large formulas quickly, which is needed to verify assertions and prune infeasible paths.

While the techniques have improved since the 70s, the fundamental ideas are still the same. The key ingredients of symbolic execution are *symbolic inputs* and *path constraints*. Symbolic inputs are special variables that stand for any value of a given type. *Symbolic values* are partially evaluated expressions, represented as abstract syntax trees, that can contain symbolic inputs. Under symbolic execution, variables and memory locations hold symbolic values instead of concrete values. Whenever calculations are performed on symbolic values, their abstract syntax trees are combined, subject to constant folding. Whenever control flow branches on an expression involving symbolic inputs, symbolic execution splits control flow into two, and follows both branches, remembering the condition needed to be true for each branch to be followed. This makes symbolic execution a graph traversal, where nodes are pairs of program states and path constraints. The possible executions with their states and path constraints form a directed graph. The path constraint accompanying a state consists of the conjunction of all conditions that must be true for this state to be reached.

Symbolic execution suffers from state-space explosion, because the number of possible paths through a program grows exponentially with the number of branch points. This means that the right search strategy is crucial for quickly finding failing assertions or programming errors. Having a good heuristics that determines which unexplored path should be followed next is essential for quickly finding interesting paths and not execute the same statements over and over. Different traversal strategies like depth-first or breadth-first are possible. Experience shows that coverage-guided breadth-first traversal is useful for finding failing assertions quickly [Cadar et al., 2008]. Coverage-guidance prioritizes paths where the next statement has not been executed often.

The path constraints of symbolic execution and the pre- and postconditions of Hoare logic look similar, and one might wonder how they relate to each other. Hoare logic and symbolic execution can be used for similar purposes, but Hoare logic is more powerful. On the one hand, like symbolic execution, Hoare logic can be used to prove that an assertion never fails, or give a counterexample if it does. However, using Hoare logic is a manual process that requires considerable effort from the programmer, especially to come up with loop invariants. Symbolic execution is automatic and just needs a powerful computer to run for extended periods of time. On the other hand, Hoare logic is more powerful than symbolic execution. Especially the option to work with loop invariants lets programmers prove properties that are not possible to prove with symbolic execution. For example, using Hoare logic it is possible to prove that an *algorithm* for the greatest common divisor (gcd)

actually computes the gcd *function*.

In this thesis we use symbolic execution for two different purposes. In chapter 6, we use it to prove that the final value of a program satisfies a given predicate. We make use of the fact that every path is accompanied by its path constraint, so every final value and its path constraint, together with the user-defined predicate, are sent to an SMT solver for verification. In essence, we allow a single assertion at the end of the program, which is checked by the symbolic execution engine. In chapter 4 we use the fact that symbolic execution explores all possible paths through a program to derive a comprehensive summary about all possible energy behaviours of the program.

1.3.6 Automated Planning and Scheduling

Planning and scheduling are two different problems that often come hand-in-hand and are sometimes mixed up. Planning is the activity of finding out how to modify the world to go from an undesirable state of affairs to a desirable one. A *planning problem* consists of a start state, a set of state-modifying actions, and a predicate on states that describes goal states. We want to find a sequence of actions that transforms the start state into a goal state. This thesis does not cover planning. Once we know *what* to do, scheduling is the activity of finding out *who* should do it and *when*. A *scheduling problem* consists of a partially ordered set of actions that all require resources, and a set of available resources. We want to find start times for all actions, and an assignment of resources to actions, subject to some optimization criteria. Optimization criteria include for example shortest time to finish, or lowest cost. A third problem, related to scheduling but slightly different, is the *assignment problem*. The assignment problem disregards task start times and focuses only on assigning resources to tasks such that quality is maximized or cost is minimized.

The problem of scheduling is as old as civilization itself. Systematic study of schedule optimization precedes computers and seems to date back to at least 1896, when the Polish economist Karol Adamiecki developed a method for *work harmonization*. By 1912, several years before Gantt charts were invented, fully developed bar charts were in use in Germany. When mainframe computers started to find their way into big companies, one of their first tasks was to optimize schedules, and with the widespread adoption of personal computers, automated scheduling was in the grasp of everyone [Weaver, 2006].

Schedule optimization has been studied extensively, due to its importance to human society. There are so many different variants of scheduling problems and associated algorithms that their categorization is almost a field of research in itself. There are specialized and general techniques, where specialized techniques are tailored to specific subsets of problems and use domain knowledge about the instance being solved to give better results, faster. Generalized techniques are applicable to a wide variety of different variants of scheduling problems, usually at the cost of speed. Heuristic methods trade completeness for speed, which means they are not guaranteed to give optimal results, but only approximations or local optima.

One popular class of scheduling algorithms are meta-heuristics. Meta-heuristics are optimization algorithms that are parametrised with fitness functions that give a score to each element of the solution space. Meta-heuristics do not impose any requirements on the fitness function, which is both an advantage and a disadvantage. The disadvantage is that the fitness function cannot be used to guide the search, which means that a given

good solution cannot be purposefully improved. Fitness functions can only be used to compare two given solutions. The advantage is that this allows them to work with non-linear, non-continuous fitness functions. This flexibility makes meta-heuristics easily applicable to a very wide variety of optimization problems, scheduling being only one of them. Meta-heuristics are not guaranteed to yield optimal solutions, only approximations. Some popular meta-heuristics are simulated annealing, ant colony optimization, and genetic algorithms.

In chapters 7 and 8 we study an assignment- and scheduling problem that arises on board of ships of the Royal Netherlands Navy. What started as a side note, a building block of an envisioned integrated mission management system, became a project in itself, due to the complexity of the matter. A lot of time went into identifying the exact problem we wanted to solve, and in finding similar work in the field of automated scheduling. We decided to use a genetic algorithm, because it was easy to formulate our problem for it, and because we wanted to allow non-linear, non-continuous, Turing-complete fitness functions. Furthermore, the result of a genetic algorithm is a whole population of results, the best it has found. This fulfilled our requirement of not only providing one good solution, but also some good alternatives.

1.4 Thesis Outline

This section lists the chapters of this thesis, and describes how they relate to the publications on which they are based.

Chapter 2: Predicting Resource Consumption of Higher-Order Workflows

This chapter is based on the paper *Predicting Resource Consumption of Higher-Order Workflows* [Klinik, Hage, Jansen, and Plasmeijer, 2017a]. We investigate how the structure of an iTasks-like program can be exploited to derive an upper bound for its overall resource consumption, given that resource requirements for basic tasks are known. For this we present a static program analysis in the form of an annotated type system. We distinguish between two categories of resources, *consumables* and *reusables*, which behave differently with respect to the various task combinators. Any resource that falls into one of the two categories can be tracked by our system. The annotated type system produces a set of constraints, which our system solves by fixpoint iteration. The result of the analysis is a vector of numbers, indicating the worst-case resource consumption for each of the resources used by the workflow.

All of the work for this paper was done by me, including literature studies on annotated type systems and type-and-effect systems, the design and implementation of the algorithm, and writing and presenting the paper. The paper was published at the *Workshop on Partial Evaluation and Program Manipulation* (PEPM) 2017 in Paris, France.

Chapter 3: The Sky is the Limit: Analysing Resource Consumption Over Time Using Skylines

We extend the resource analysis of chapter 2 to resource consumption over time. This chapter is based on the paper *The Sky is the Limit: Analysing Resource Consumption Over Time Using Skylines* [Klinik, Jansen, and Plasmeijer, 2017b]. Basic tasks are annotated

with a duration, in addition to resource requirements. This allows estimating resource consumption over time in graphs we call *skylines*. The key aspects of this work are operators to concatenate, merge, and add skylines, respectively for consumable and reusable resources. The operators are all monotone with respect to skyline ordering. This allows the same fixpoint iteration algorithm from chapter 2 to be used to solve constraints. The result of the analysis is a vector of skylines, one for each resource, indicating the worst-case resource consumption graph for each of the resources used by the workflow.

All of this chapter is my own contribution. The paper was published at the *Symposium on Implementation and Application of Functional Languages* (IFL) 2017 in Bristol, UK.

Chapter 4: Skylines for Symbolic Energy Consumption Analysis

We combine two lines of work, the skylines from chapter 3 and the energy consumption analysis by van Gastel [2016]. This chapter is based on the paper *Skylines for Symbolic Energy Consumption Analysis* [Klinik, van Gastel, Kop, and van Eekelen, 2020]. The analyzed language is an imperative C-like language instead of the iTasks-like languages of the previous chapters. The only tracked resource is power draw. Changes in power draw originate from *component calls*, which are special statements that control hardware. The system uses symbolic execution, which is a model-checking technique and thus gives precise results, as opposed to the overapproximation of the previous chapters. The increased precision comes at the cost of higher computational complexity. The result of the analysis is a set of energy skylines, one for each possible execution path through a program, together with path constraints expressing bounds on user input required to reach each path. We present a skyline merge algorithm that condenses all result skylines into a neat summary.

I designed most of the system, and did all of its implementation. The implementation is comprised of the parser, the concrete and symbolic execution engines, the merge algorithm, and the diagram generator. Furthermore I wrote the sections on syntax, semantics, and symbolic execution of the paper. The paper was published at the *25th International Conference on Formal Methods for Industrial Critical Systems* (FMICS) 2020, held online.

The content of section 4.9 did not make it into the paper, and was made available online in an informal techreport. I designed all the described semantics, and wrote all of this section.

Chapter 5: TopHat: A Formal Foundation for Task-Oriented Programming

This chapter is based on the paper *TopHat: A Formal Foundation for Task-Oriented Programming* [Steenvoorden, Naus, and Klinik, 2019]. Our goal was to identify the essence of task-oriented programming, and create a task-oriented programming language small enough to be susceptible to formal reasoning. We present TopHat, which is a simply-typed lambda calculus with references and constants for task-oriented programming. We define the semantics of the language in three layers, one for the underlying host language, one for the non-interactive task constructs, and one for the interactive ones. We provide proofs for progress and preservation. There is an implementation of the semantics in Haskell.

My contributions to this paper start with considerable involvement in the work leading up to the paper. We have conducted interviews with many staff members who have been participating in the development of iTasks in the past decade, to collect different opinions about the essence of task-oriented programming. I performed literature studies

on functional reactive programming, concurrent programming, and process algebras, to get an understanding of how they relate to task-oriented programming. Based on these preparations, I was involved in the design of TopHat, which took multiple iterations until it reached the form found in the paper. I wrote most of the introductory section of the paper, most of the examples, and large parts of the related work and future work sections. Some of the sections and examples I wrote, especially the comparison of process algebras and TOP, did not make it into the paper due to the page limitation. They have been restored for this chapter. The paper was published at the *International Symposium on Principles and Practice of Declarative Programming* (PPDP) 2019 in Porto, Portugal.

Chapter 6: A Symbolic Execution Semantics for TopHat

Where we have paved the way in chapter 5, we now deliver on our promise to subject TopHat programs to formal reasoning. This chapter is based on the paper *A Symbolic Execution Semantics for TopHat* [Naus, Steenvoorden, and Klinik, 2019]. We replace the labels of the labelled transition system for the interactive task constructs of chapter 5 with symbolic inputs. When user input is required to advance execution of a term, the symbolic execution semantics generates a fresh symbolic input. When the semantics encounters a conditional that depends on symbolic input, both branches are executed independently, and the condition that led to each branch is remembered as a first-order boolean formula, the path constraint. The result of symbolic execution is a set of final task values, accompanied with the path constraints that led to them. The user can specify a property about final task values, and the system matches it with each path constraint, thereby proving whether all possible outcomes of a program fulfil the property.

My contributions to this chapter include literature studies on symbolic execution techniques and axiomatic program verification, especially with regard to functional programming, and bringing this knowledge into the design of the system. For the paper, I wrote the section on related work, and most of the examples. The paper was published at the *Symposium on Implementation and Application of Functional Languages* (IFL) 2019 in Singapore.

Chapter 7: Dynamic Resource and Task Management

This chapter is based on the paper *Dynamic Resource and Task Management* [Klinik, Jansen, and Bolderheij, 2018]. The paper was written in collaboration with the Netherlands Defence Academy, within the scope of the Manning and Automation project of the Royal Netherlands Navy. We study opportunities and conditions for how automated scheduling can be applied to the workflows on board of navy ships. We conclude that we have to solve an assignment problem and a scheduling problem simultaneously. The quality measure for the assignments should be highly flexible, and depend on all kinds of factors like outside circumstances, the state of the ship, and other assignments. We further discuss how such a scheduler could be integrated into a mission management framework, keeping the human in control should the need for overriding decisions arise.

This paper came forth from my attending the biannual Manning and Automation project workshops, two military trade shows, and weekly interviews and brainstorming sessions with personnel from the Netherlands Navy and Defence Academy. All of this chapter is my own contribution. The paper was published in the book *Netherlands Annual Review of*

Military Studies (NL ARMS) 2018. It received examination by the editors of the book, but no peer review.

Chapter 8: Resource Scheduling with Computable Constraints for Maritime Command and Control

This chapter is based on the paper *Resource Scheduling with Computable Constraints for Maritime Command and Control* [Klinik, Jansen, and Plasmeijer, 2019]. In this chapter we implement a scheduler according to the requirements identified in chapter 7. We precisely define the kind of scheduling problem we want to solve, compare it with problems found in the literature on automated scheduling, and describe a genetic algorithm that computes assignments and schedules.

All of this chapter is my own contribution. This includes literature studies on automated planning and scheduling, multi-criteria decision making, genetic algorithms, and combinatorial optimization. I implemented the scheduler and presented it at our booth at the Manning and Automation symposium 2018 in Den Helder, the Netherlands. The paper was published at the conference *Maritime/Air Systems & Technologies* (MAST Asia) 2019 in Tokyo, Japan. It received examination by the conference committee, but no peer review.

Part I

Resource Analysis

2 Predicting Resource Consumption of Higher-Order Workflows

We present a type and effect system for the static analysis of programs written in a simplified version of iTasks. iTasks is a workflow specification language embedded in Clean, a general-purpose functional programming language. Given costs for basic tasks, our analysis calculates an upper bound of the total cost of a workflow. The analysis has to deal with the domain-specific features of iTasks, in particular parallel and sequential composition of tasks, as well as the general-purpose features of Clean, in particular let-polymorphism, higher-order functions, recursion and lazy evaluation. Costs are vectors of natural numbers where every element represents some resource, either consumable or reusable.

2.1 Introduction

Workflows are algorithmic descriptions of how to combine basic tasks in order to achieve some higher-level goal. The reasons to employ workflows are to understand, define and streamline complicated processes, usually involving many contributing performers. In other words, it is all about optimizing costs.

The most popular workflow systems come in the flavour of box and arrow diagrams. Most notably there is BPMN [Object Management Group, 2009], the Business Process Model and Notation, which is the de-facto industry standard with dozens if not hundreds of software tools from different manufacturers for designing and simulating workflows. From the perspective of programming language design, BPMN resembles an imperative programming language with GOTOs and barely any capability for abstraction. Control flow is explicit by means of arrows between boxes, but data flow is implicit.

In contrast, task-oriented programming (TOP) is a programming paradigm that allows specifying workflows in a declarative way. TOP programs are distributed applications where users work together on the internet. They are built using the four concepts *tasks*, *shared data*, *generic interaction*, and *task combinators*. There is an implementation of TOP called iTasks [Plasmeijer et al., 2012], which comes as an embedded domain-specific language in the pure, lazy functional programming language Clean. As such, workflows written in iTasks can make use of mechanisms common to functional programming, such as higher-order functions, polymorphic static typing, algebraic datatypes and pattern matching. Furthermore, data flow in iTasks is explicit, as tasks can return values. In particular, tasks can take other tasks as input or return them as values, making workflows in iTasks higher-order. A demonstration of the capabilities of iTasks can be found in Lijnse et al. [2012], where it has been used to write a crisis management tool for the Dutch coast guard.

In this chapter we present a static analysis for iTasks-like programs that, given costs for basic tasks, calculates the total cost of a compound workflow. The analysis comes

in the form of a type and effect system with polymorphism, polyvariance and subtyping. The analysis is parametrised in the types of resources a workflow uses, which means programmers can define arbitrary units of costs for their workflows.

2.1.1 Motivating Example

Let us look at an example to get an impression what the analysis can do. Consider the following program.

$$
\begin{aligned}
&\textbf{let } \mathit{forever} = \textbf{fix}\, f\, x.x \gg f\, x \textbf{ in}\\
&\textbf{let } \mathit{approve} = \textbf{fn}\, t.\textbf{use}\ [1\ \text{Supervisor}]\ \textbf{True in}\\
&\textbf{let } \mathit{approved} = \textbf{fn}\, t.\mathit{approve}\ t \gg\!= \textbf{fn}\, \mathit{ok}.\\
&\quad \textbf{if } \mathit{ok}\ \textbf{then } t\ \textbf{else } (\textbf{return } 0)\ \textbf{in}\\
&\textbf{let } \mathit{initialize} = \textbf{use}\ [3\ \text{Toolboxes}]\ 0\ \textbf{in}\\
&\textbf{let } \mathit{work} = \textbf{use}\ [1\ \text{lFuel} + 1\ \text{Toolboxes}]\ 0\ \textbf{in}\\
&\mathit{approved}\ \mathit{initialize} \gg \mathit{approved}\ (\mathit{forever}\ \mathit{work})
\end{aligned}
$$

The function *forever* executes its argument task in an endless loop. Approval of a task is represented by *approve*. In a real iTasks program the task t could be presented on screen to a person who then decides whether it should be executed. In this example, tasks are always approved. The function *approved* only executes its argument if it gets approval. The tasks *initialize* and *work* are stand-ins for some tasks where real work happens. They cost 3 toolboxes, and 1 litre of fuel and a toolbox respectively. The main expression first runs initialization and then the actual work in an infinite loop, but only after asking for approval. The analysis, when run with this program as input, computes as answer the cost [1 Supervisor + ∞ lFuel + 3 Toolboxes]. This is because in each iteration of the loop more fuel is consumed, while the toolboxes can be reused. How exactly the analysis computes this answer is the subject of this chapter.

The rest of this chapter is organized as follows. In section 2.2 we define syntax and operational semantics of a language to specify workflows. In section 2.3 we develop the annotated type system that performs the cost analysis. Section 2.4 describes an algorithm that computes cost estimations. Section 2.5 discusses the capabilities of the method by means of examples.

2.2 Syntax and Semantics

In this section we define the syntax and operational semantics of a programming language to specify workflows. We consider two sorts of resources, *consumables* and *reusables*. Consumable resources are used up when a task that requires them is executed. Reusable resources become available again upon completion, and are claimed exclusively during execution of a task. We assume that it is implicitly understood which resources are consumable and which are reusable.

$$
\begin{aligned}
e ::= {} & b \mid i \mid () \mid x \mid \mathbf{fn}\, x.e \mid \mathbf{fix} f x.e \mid e_1 e_2 \\
& \mid \mathbf{if}\, e_c\, \mathbf{then}\, e_t\, \mathbf{else}\, e_e \mid \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \mid e_1 \odot e_2 \\
& \mid \mathbf{use}\, [k]\, e \mid \mathbf{return}\, e \\
& \mid e_1 \,\&\, e_2 \mid e_1 \gg\!= e_2 \mid e_1 \gg e_2 \\
k ::= {} & nu \mid nu + k
\end{aligned}
$$

Figure 2.1: Syntax of the language

2.2.1 A Programming Language for Workflows

Our language is a simplified version of Clean and iTasks. It is a small functional programming language with higher-order functions, non-recursive let-bindings and a fixpoint combinator. Tasks and workflows exist as domain-specific constants and combinators in the language. The syntax of the language is defined in fig. 2.1 The general-purpose part of the language has Boolean and integer constants b and i, program variables x, abstraction, application, if-then-else and let-bindings. The symbol $\odot$ stands for the usual binary operators for arithmetic, Boolean connectives and comparison. There is a fixpoint combinator $\mathbf{fix} f x.e$ that defines recursive functions with one argument.

Tasks in our language always yield values, but not all real-world tasks do. In monadic programming one usually uses the unit type when values do not matter. The unit type and its value value are both denoted by $()$.

The domain-specific part of the language has a primitive **use** for basic tasks, and task combinators for sequential and parallel composition. All basic tasks are represented by the **use** operator, where k denotes the cost of executing the task. Costs are given in a polynomial-like syntax where n is a natural number and u the unit of a resource. For example $\mathbf{use}\ [2S + 3B]\ 5$ may denote a task that needs 2 screwdrivers, 3 bottles of wine, and yields the value 5. Costs and resources are discussed in more detail in section 2.2.2. Expressions of the form **return** e are tasks that have been executed and yield value e.

There are three combinators for tasks, the parallel combinator (&) and two variants of sequential composition. The regular *bind* operator ($\gg\!=$) executes its left argument first and passes the resulting value to its right argument as usual. The *sequence* operator ($\gg$) ignores the value of its left argument and yields the value of its right argument. The sequence operator is useful for example programs where the values of tasks do not matter. We include it in our language for convenience, being well aware that it can easily be defined in terms of bind. Since $\gg$ is a variant of $\gg\!=$ with identical cost behaviour, we ignore it in the formal part of this chapter but still use it in examples.

The parallel combinator executes both its arguments simultaneously. In iTasks the result value of parallel composition is a tuple containing the values of both tasks. Our language does not have tuples, a deliberate decision because we want to focus more on side effects than on values. Adding tuples would inflate the various definitions relating to our language while not providing substantial new insight regarding cost analysis. We take the liberty of bending the semantics of the parallel combinator a bit to avoid tuples, saving some space in this chapter. Both arguments to (&) and its result are of type *task* ().

2.2.2 A Domain for Representing Costs

To define the semantics of the language, we need to make the notions of resource and cost precise. We define a *resource* to be a positive integral quantity, possibly infinite. Every resource has a unit, so we are not just talking about numbers but about quantities of certain things. Think of litres of fuel or numbers of first-aid kits.

Definition 2.2.1. (Extended natural numbers) The extended natural numbers are the natural numbers with infinity: $\overline{\mathbb{N}} = \{ 0, 1, 2, \ldots, \infty \}$. The only operations we need are addition and comparison, which are extended with ∞ in the natural way.

Definition 2.2.2. (Cost) Let U be a finite set. A *cost over* U is a function $\gamma : U \to \overline{\mathbb{N}}$. Usually, U is understood from the context and we just talk about *costs*. □

The set U contains the units of the resources of interest in a given workflow. A cost can be seen as a U-indexed set of numbers. For example, let $U = \{ mW, l \text{ water}/h, \text{ Potatoes} \}$. In a situation like that we are talking about tasks that require any number of these resources. We can then express that a task uses 5 litres of water per hour, 2 potatoes, and no power by associating the following cost to it $[mW \mapsto 0,\ l \text{ water}/h \mapsto 5,\ \text{Potatoes} \mapsto 2]$. We shall adopt a polynomial-like notation $0mW + 5l/h + 2P$ for this. Examples in this chapter do not mention concrete resources. We are interested in the abstract idea of consumables and reusables, and will use C and R as stand-ins for some consumable and reusable resource respectively. Our example tasks will have costs like $5C + 3R$.

The analysis is based on generating and solving constraints. Constraint solving relies on Tarski's fixpoint theorem, which states that every monotone function on a complete lattice has a least fixpoint. In anticipation of that, the domain of costs must be a complete lattice. We first define the lattice of a single resource.

Lemma 2.2.3. *The structure* $(\overline{\mathbb{N}}, \leq, \sqcup, \sqcap)$ *is a complete lattice. Joins are maxima, meets are minima. The top element is* ∞*, the bottom element is* 0*.* □

As said in definition 2.2.2, a cost is a collection of such numerical quantities. The order on costs comes from the order of the constituents.

Definition 2.2.4. The set of all possible costs is defined as the function space from U to $\overline{\mathbb{N}}$, extended with a bottom element, written as $[U \to \overline{\mathbb{N}}]_\perp$. □

Definition 2.2.5. Costs are ordered pointwise. That is, for two costs $\delta, \gamma \in [U \to \overline{\mathbb{N}}]_\perp$

$$\delta \sqsubseteq \gamma \iff \begin{cases} \delta = \perp & \text{or} \\ \forall u \in U. \delta(u) \leq \gamma(u) & \text{if } \delta, \gamma \neq \perp \end{cases}$$ □

In this chapter we define several binary operators on costs. All of them are non-strict. This means for all $op \in \{ \sqcup, +, \boxplus, \nabla \}$ we implicitly have:

$$\gamma \mathbin{op} \delta = \begin{cases} \gamma & \text{if } \delta = \perp \\ \delta & \text{if } \gamma = \perp \end{cases}$$

The definitions of these operators will cover the cases where both γ and $\delta \neq \perp$.

Definition 2.2.6. Joins of costs are defined pointwise. That is, for two costs δ and γ,

$$\delta \sqcup \gamma = \lambda u.\delta(u) \sqcup \gamma(u).$$

Meets of costs are defined analogously. □

In this chapter we work with lattices whose relation we denote by $\sqsubseteq$. This symbol should be read as "below", which implicitly includes "or equal". The inverse relation $\sqsupseteq$ should be read as "above". Similarly, when comparing tasks the terms "cheaper" and "more expensive" implicitly include possible equality.

Lemma 2.2.7. *Costs form a complete lattice* $\mathbb{D} = ([U \to \overline{\mathbb{N}}]_\bot, \sqsubseteq, \sqcup, \sqcap)$. *The top element is* $\lambda u.\infty$, *the bottom element is the artificial* $\bot$. *Joins are pointwise maxima, meets are pointwise minima.* □

This structure will serve as the domain for both the operational semantics and the static analysis. This domain is called $\mathbb{D}$. We use the letters γ and δ to denote elements of $\mathbb{D}$. The fact that $\mathbb{D}$ is a complete lattice will allow us to use Tarski's fixpoint theorem for solving constraint sets in lemma 2.3.5.

We define three binary operations on $\mathbb{D}$ which we use to model sequential composition, parallel composition, and recursion.

Definition 2.2.8. Addition of costs is defined pointwise.

$$\gamma + \delta = \lambda u.\gamma(u) + \delta(u)$$ □

Definition 2.2.9. The *combine* operator $\boxplus$ for costs is defined pointwise as the maximum of reusables and sum of consumables.

$$\gamma \boxplus \delta = \lambda u.\begin{cases} \gamma(u) \sqcup \delta(u) & \text{if } u \text{ is reusable} \\ \gamma(u) + \delta(u) & \text{if } u \text{ is consumable} \end{cases}$$ □

The combine operator implements the idea that consumables are used up but reusables can be reused. If the left argument already requires 5 units of a reusable resource R, but the right argument only requires 3, then the total requirement is $5R$.

Definition 2.2.10. Let γ, δ be costs. The *widening of* γ *by* δ, written $\gamma \nabla \delta$, is defined as follows.

$$\gamma \nabla \delta = \lambda u.\begin{cases} \infty & \text{if } \delta(u) > \gamma(u) \\ \gamma(u) & \text{otherwise} \end{cases}$$ □

We use widening to guarantee termination of fixpoint iteration when the analysed program contains recursion. The asymmetric definition of widening is intentional. The left argument will always be the overall cost of a recursion and the right argument will always be the cost of a single pass. When the cost of a single pass exceeds the cost of the recursion overall, which means that each iteration adds to the cost, the widening operator yields infinity. Widening is used in the implementation in section 2.4 and discussed in more detail in section 2.5.

2.2.3 An Operational Semantics

We split the operational semantics in two parts. Both are small-step structural operational semantics. The general-purpose part, denoted by a normal arrow $\rightarrow$, applies to non-task expressions. The domain-specific part of the semantics applies to task expressions. It is denoted by a two-headed arrow $\twoheadrightarrow$ to emphasize its relation with the bind operator $\gg=$.

The general-purpose semantics rewrites expressions with judgements of the form $Expr \rightarrow Expr$. The domain-specific semantics $Expr \times \mathbb{D} \twoheadrightarrow Expr \times \mathbb{D}$ rewrites expressions while recording the resources that are used during reduction. The names of the rules of the semantics are prefixed by gs- and ds-, which are to be read as "general-purpose step" and "domain-specific step". The split into two semantics comes from the fact that we are dealing with two languages: a domain-specific task language embedded in a functional host language. Task expressions are values for the host semantics, which ensures that resources are only consumed when the task semantics makes a step. This gives the same cost behaviour for both call-by-name and call-by-value host languages. Our choice for a call-by-name host language is arbitrary, motivated by the fact that Clean is lazy. The setup is very similar to the treatment of IO in Haskell by Peyton-Jones [2001].

To define a small-step semantics for the parallel composition of tasks, we add a new syntactic form to the language, called *process pool*. Process pools are part of the abstract syntax, but programmers cannot use them directly. They only exist temporarily during reduction and disappear once both tasks have been executed. When the parallel composition of two tasks needs to be reduced, a process pool springs into existence and keeps track of the costs of the tasks separately, so that no sharing of resources takes place. Process pools hold exactly two tasks, but can be nested.

$$e ::= \ldots \mid pp(e_1 \,\&\, e_2, \gamma_1, \gamma_2)$$

In a process pool $e_1 \,\&\, e_2$ are two tasks in progress of being executed and γ_1 and γ_2 are their respective costs.

The general-purpose semantics is given in fig. 2.2.

[gs-fix] This rule states that a fixpoint reduces to a function where every occurrence of f in the function body is replaced by the fixpoint itself.

[gs-app-cong] This rule states that in an application, the expression in function position evaluates to a value first.

[gs-app] This rule implements call-by-name reduction. The argument is not reduced but substituted as-is.

[gs-let] This rule implements let-bindings in a call-by-name style. The duplication of costs which results from substituting a task into an expression at several places is intended, because executing a task multiple times costs more.

[gs-if-cong] This rule states that in a conditional, the condition is evaluated first.

[gs-if-t], [gs-if-f] These rules reduce to the respective branch if the condition is **True** or **False**.

The domain-specific semantics is given in fig. 2.3.

[ds-use] This rule stands for the execution of a basic task in a call-by-name style. On the left-hand side, γ is the cost of the program so far. Executing the task **use** $[k]\ e$ uses up

$$\textbf{[gs-fix]} \quad \textbf{fix} f x.e \to \textbf{fn}\, x.e[f \mapsto \textbf{fix} f x.e]$$

$$\textbf{[gs-app-cong]} \ \frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2}$$

$$\textbf{[gs-app]} \quad (\textbf{fn}\, x.e_b) e_x \to e_b[x \mapsto e_x]$$

$$\textbf{[gs-let]} \quad \textbf{let}\ x = e_x\ \textbf{in}\ e_b \to e_b[x \mapsto e_x]$$

$$\textbf{[gs-if-cong]} \ \frac{e_c \to e_c'}{\textbf{if}\ e_c\ \textbf{then}\ e_t\ \textbf{else}\ e_e \to \textbf{if}\ e_c'\ \textbf{then}\ e_t\ \textbf{else}\ e_e}$$

$$\textbf{[gs-if-t]} \quad \textbf{if True then}\ e_t\ \textbf{else}\ e_e \to e_t$$

$$\textbf{[gs-if-f]} \quad \textbf{if False then}\ e_t\ \textbf{else}\ e_e \to e_e$$

Figure 2.2: Small-step semantics for general-purpose expressions

the resources k, which is modelled by adding k to γ on the right-hand side. The result is **return** e, which indicates that the task has been executed and yields value e.

[ds-pure] This rule rule lifts the general-purpose semantics into the domain-specific semantics. It is needed for expressions of type *task* that must make a normal step. For example:

$$\langle \textbf{if True then}\ (\textbf{use}\ [k]\ 5)\ \textbf{else}\ (\textbf{use}\ [l]\ 7), \gamma \rangle \twoheadrightarrow$$
$$\langle \textbf{use}\ [k]\ 5, \gamma \rangle$$

[ds-bind-ret] This rule implements the bind operator. If the left argument of the bind is a task that has been executed, then its result value is passed as an argument to the left argument of the bind.

[ds-bind-cong] These rules state that the left argument of a bind must be reduced first. The cost of the step of e_1 is added to the overall cost of the program.

[ds-para-init] This rule creates a process pool in which the parallel composition is executed. When both tasks are fully reduced, which means they are of the form **return** e, **[ds-para-ret]** calculates the resulting cost such that no resources are shared between the parallel tasks. The result value is the value is always `()`, a simplification that, as discussed earlier, saves introduction of tuples to our type system at the cost of deviating from how iTasks works.

[ds-para-cong-l], [ds-para-cong-r] The rules state how subexpressions must be evaluated so that eventually **[ds-para-ret]** applies.

$$\textbf{[ds-use]} \quad \langle \textbf{use}\ [k]\ e, \gamma \rangle \twoheadrightarrow \langle \textbf{return}\ e, \gamma \boxplus k \rangle$$

$$\textbf{[ds-pure]}\ \frac{e \to e'}{\langle e, \gamma \rangle \twoheadrightarrow \langle e', \gamma \rangle}$$

$$\textbf{[ds-bind-ret]} \quad \langle \textbf{return}\ e_1 \gg\!= e_2, \gamma \rangle \twoheadrightarrow \langle e_2 e_1, \gamma \rangle$$

$$\textbf{[ds-bind-cong]}\ \frac{\langle e_1, \gamma \rangle \twoheadrightarrow \langle e_1', \gamma' \rangle}{\langle e_1 \gg\!= e_2, \gamma \rangle \twoheadrightarrow \langle e_1' \gg\!= e_2, \gamma' \rangle}$$

$$\textbf{[ds-para-init]} \quad \langle e_1\ \&\ e_2, \gamma \rangle \twoheadrightarrow \langle pp(e_1\ \&\ e_2, \bot, \bot), \gamma \rangle$$

$$\textbf{[ds-para-ret]}\ \langle pp(\textbf{return}\ ()\ \&\ \textbf{return}\ (), \gamma_1, \gamma_2), \gamma \rangle \twoheadrightarrow \langle \textbf{return}\ (), \gamma \boxplus (\gamma_1 + \gamma_2) \rangle$$

$$\textbf{[ds-para-cong-l]}\ \frac{\langle e_1, \gamma_1 \rangle \twoheadrightarrow \langle e_1', \gamma_1' \rangle}{\langle pp(e_1\ \&\ e_2, \gamma_1, \gamma_2), \gamma \rangle \twoheadrightarrow \langle pp(e_1'\ \&\ e_2, \gamma_1', \gamma_2), \gamma \rangle}$$

$$\textbf{[ds-para-cong-r]}\ \frac{\langle e_2, \gamma_2 \rangle \twoheadrightarrow \langle e_2', \gamma_2' \rangle}{\langle pp(e_1\ \&\ e_2, \gamma_1, \gamma_2), \gamma \rangle \twoheadrightarrow \langle pp(e_1\ \&\ e_2', \gamma_1, \gamma_2'), \gamma \rangle}$$

Figure 2.3: Small-step semantics for task expressions

2.3 The Analysis

In this section we present a static analysis that estimates the cost of executing a workflow. The estimation is conservative, which means that no execution of a workflow uses more resources than the analysis predicts. The analysis is realized as a type and effect system with polymorphism, polyvariance and subtyping. The type system collects constraints such that a solution of the constrains gives an estimation of the cost of executing the program.

2.3.1 The Annotated Type System

The type system has type variables α, three base types for Booleans, integers, and unit, and two type constructors for functions and tasks. The only annotated types are tasks, where annotations come in the form of annotation variables β. Annotation variables will be given meaning by the solution of constraint sets. Types are formed by the following grammar.

$$\hat{\tau} ::= \alpha \mid \mathit{bool} \mid \mathit{int} \mid () \mid \mathit{task}\ \beta\ \hat{\tau} \mid \hat{\tau}_1 \to \hat{\tau}_2$$

To improve the precision of the analysis, we make use of polymorphism, polyvariance, and subtyping. These three techniques allow different types, and therefore different costs, for the same variable in different contexts. Though not strictly necessary for the system to be sound, they increase its precision by preventing poisoning in the following situations. Polymorphism, as in normal Hindley-Milner systems, allows different occurrences of a

variable in the body of a let binding to have different types, provided that the type can be generalized. As types can be chosen to fit the context, so can be the costs. Polyvariance is a weak form of polymorphism. It is applicable to let-bound variables, and comes into play when the type of a variable cannot be generalized. Polyvariance allows the cost of a task to be increased when a context requires it, without affecting the variable's type in other contexts. Subtyping is useful for lambda-bound variables, which are not subject to polymorphism and polyvariance. It also allows the cost of a task to be increased when a context requires it without affecting the type in other contexts. Subtyping comes into play in branches of conditionals and the arguments of function application.

The type system uses *type schemes* for polymorphism and polyvariance. Our type schemes are *qualified*, which means they include constraints relating bound variables. Type schemes are formed by the following grammar.

$$\sigma ::= \hat{\tau} \mid \forall \vec{\alpha}\, \vec{\beta}.C \Rightarrow \hat{\tau}$$

In a type scheme, $\vec{\alpha}$ are bound type variables, $\vec{\beta}$ bound annotation variables, and C is a set of constraints that restricts bound annotation and type variables. When $\vec{\alpha}$ and $\vec{\beta}$ are empty, which means there are no bound variables in $\hat{\tau}$, C will usually be empty as well. In this case we just write $\hat{\tau}$ instead of $\forall().\emptyset \Rightarrow \hat{\tau}$.

Definition 2.3.1. (Free type and annotation variables) We define two functions on types and type schemes, *FTV* and *FAV*. *FTV* returns all free type variables of a type scheme, and *FAV* all free annotation variables. They are defined by induction on the syntax of types. We omit the formal definition. For example

$$FAV(\forall \alpha_1 \beta_1.\{\, \beta_2 \sqsupseteq \beta_1 \,\} \Rightarrow task\, \beta_1\, \alpha_1 \to task\, \beta_2\, \alpha_2) = \{\, \beta_2 \,\}.$$

These functions extend pointwise to environments. We also define them for annotations and constraint sets, where they just return all occurring variables, because there are no bound variables in constraints. □

Definition 2.3.2. (Annotations) Annotations are expressions denoting costs. They are used in constraints to describe the cost behaviour of programs. Annotations are formed by the following grammar.

$$\varphi ::= k \mid \beta \mid \varphi_1 + \varphi_2 \mid \varphi_1 \boxplus \varphi_2 \mid \varphi_1 \nabla \varphi_2$$

k stands for cost constants like $[2C + 7R]$. β stands for annotation variables. The three operators $+, \boxplus, \nabla$ stand for their respective operations on $\mathbb{D}$, defined in section 2.2.2. □

Definition 2.3.3. (Constraints) Constraints come in two forms, one that relates types and one that relates costs. Constraints are formed by the following grammar.

$$c ::= \hat{\tau}_1 <: \hat{\tau}_2 \mid \beta \sqsupseteq \varphi$$

A constraint of the form $\hat{\tau}_1 <: \hat{\tau}_2$ is called a *subsumption constraint* and records the fact that $\hat{\tau}_1$ must be a subtype of $\hat{\tau}_2$. A constraint of the form $\beta \sqsupseteq \varphi$, called an *effect constraint*, means that the cost β is above the cost expressed by φ. □

Effect constraints $\beta \sqsupseteq \varphi$ use the symbol "above" and always have a variable on the left, and an annotation on the right-hand side. This is customary in literature on type and effect systems, and we adopt it here. Please note that the direction of subsumption constraints is the opposite of effect constraints, again an adoption of convention in the literature.

Standard Hindley-Milner type inference works by solving type equality constraints. Equality constraints can be solved as soon as they are discovered, so most algorithms perform unification on the fly, without ever explicitly generating constraint sets. We work with inequality constraints, which can only be solved once all constraints are known. This is why we have to explicitly construct and keep track of constraints during type inference, and solve them at the end in a separate phase.

Definition 2.3.4. (Annotation semantics) Given an annotation φ, we define the interpretation of φ, written $[\![\varphi]\!]$, as a function from assignments to costs. An assignment $s : \mathrm{AnnVar} \to \mathbb{D}$ is a mapping from annotation variables to costs.

$$
\begin{aligned}
[\![k]\!]s &= k \\
[\![\beta]\!]s &= s(\beta) \\
[\![\varphi_1 + \varphi_2]\!]s &= [\![\varphi_1]\!]s + [\![\varphi_2]\!]s \\
[\![\varphi_1 \boxplus \varphi_2]\!]s &= [\![\varphi_1]\!]s \boxplus [\![\varphi_2]\!]s \\
[\![\varphi_1 \bigtriangledown \varphi_2]\!]s &= [\![\varphi_1]\!]s \bigtriangledown [\![\varphi_2]\!]s
\end{aligned}
$$

Interpretations of annotations are used in the effect constraint solver in section 2.4. □

To streamline the discussion about constraints and constraint sets, we introduce the constraint entailment relation, which makes use of the following fact.

Lemma 2.3.5. *Any constraint set C gives rise to a monotone function*

$$F_C : (AnnVar_C \to \mathbb{D}) \to (AnnVar_C \to \mathbb{D}).$$ □

Monotonicity comes from the fact that the interpretation of annotations, see definition 2.3.4, is monotone. AnnVar_C is the set of annotation variables occurring in C. Our domain $\mathbb{D}$ is a complete lattice, and so is the function space $\mathrm{AnnVar}_C \to \mathbb{D}$. As such, F_C has a least fixpoint which is at the same time the least solution of C.

Definition 2.3.6. (Constraint entailment) An assignment *s satisfies* an effect constraint $\beta \sqsupseteq \varphi$ iff $s(\beta) \sqsupseteq [\![\varphi]\!]s$. Let C be a constraint set and c a constraint. We write $C \Vdash c$ if the minimal solution s of C satisfies c. The existence of such a solution is guaranteed by Tarski's fixpoint theorem and lemma 2.3.5. We write $C \Vdash D$ if every solution of C is a solution of D. Satisfaction of subsumption constraints is defined in definition 2.3.7. □

Definition 2.3.7. (Subtyping) Subtyping is the reflexive transitive closure of the relation defined by the rules in fig. 2.4. Reflexivity in particular means that $\Vdash \mathit{bool} <: \mathit{bool}$ and $\Vdash \mathit{int} <: \mathit{int}$. □

Subtyping expresses that if the types of two expressions $e_1 : \hat{\tau}_1$ and $e_2 : \hat{\tau}_2$ are in the subtype relation $\hat{\tau}_1 <: \hat{\tau}_2$, then e_1 can be used in all the places where e_2 can be used. In our system this means that either e_1 and e_2 are of base types, which means $\hat{\tau}_1 = \hat{\tau}_2$, or $\hat{\tau}_1$ is cheaper than $\hat{\tau}_2$. If a context has enough resources for $\hat{\tau}_2$, it also can deal with $\hat{\tau}_1$. This is essentially what the subtyping rule **[st-task]** says.

$$\textbf{[st-task]}\ \frac{C \Vdash \beta_1 \sqsubseteq \beta_2 \qquad C \Vdash \hat{\tau}_1 <: \hat{\tau}_2}{C \Vdash \mathit{task}\ \beta_1\ \hat{\tau}_1 <: \mathit{task}\ \beta_2\ \hat{\tau}_2} \qquad \textbf{[st-fun]}\ \frac{C \Vdash \hat{\tau}'_a <: \hat{\tau}_a \qquad C \Vdash \hat{\tau}_r <: \hat{\tau}'_r}{C \Vdash \hat{\tau}_a \to \hat{\tau}_r <: \hat{\tau}'_a \to \hat{\tau}'_r}$$

Figure 2.4: The subtyping relation

Example 2.3.8. Consider the constraint set $C = \{\ \beta_1 \supseteq [1C], \beta_2 \supseteq [2C]\ \}$. The minimal solution s of C has $[\beta_1 \mapsto [1C], \beta_2 \mapsto [2C]]$. By definition of $\Vdash$, this means $C \Vdash \beta_1 \sqsubseteq \beta_2$. This allows us to form the following small derivation, which witnesses the fact that the two task types in the conclusion are in fact in the subtype relation.

$$\textbf{[st-task]}\ \frac{C \Vdash \beta_1 \sqsubseteq \beta_2 \qquad \Vdash \mathit{int} <: \mathit{int}}{C \Vdash \mathit{task}\ \beta_1\ \mathit{int} <: \mathit{task}\ \beta_2\ \mathit{int}}$$

□

The rule **[st-fun]** implements subtyping for functions. As usual in systems with subtyping, functions are contravariant in the argument type $\hat{\tau}_a$ and covariant in the result type $\hat{\tau}_r$. Covariant means that the result types of two functions are in the same subtyping order as the functions overall, while contravariant means that the order of the argument types is reversed. To understand the latter, consider two functions f and g with the same result type $\hat{\tau}_r$, but with argument types in relation $\hat{\tau}_f <: \hat{\tau}_g$. In other words f expects cheaper arguments than g. Then, according to **[st-fun]**, $\hat{\tau}_g \to \hat{\tau}_r <: \hat{\tau}_f \to \hat{\tau}_r$, which means g can be used in all contexts in which f can be used and possibly more. The intuitive reason is that if g can deal with expensive arguments, it can also deal with the cheaper arguments of f. The formal reason is that our type system allows any expression to be considered as more expensive if needed. If the application $f\,e$ is well-typed then the type system can make e more expensive so that $g\,e$ is also well-typed.

Definition 2.3.9. (Generalization) Let Γ be an environment, C a constraint set and $\hat{\tau}$ a type. The generalization of $\hat{\tau}$ with respect to Γ and C is defined as follows.

$$\begin{aligned} &\mathit{generalize}(\hat{\tau}, C, \Gamma) = \forall \vec{\alpha}\,\vec{\beta}.C \Rightarrow \hat{\tau} \quad \mathit{where} \\ &\quad \vec{\alpha} = \mathit{FTV}(\hat{\tau}) \setminus \mathit{FTV}(\Gamma) \\ &\quad \vec{\beta} = \mathit{FAV}(\hat{\tau}) \setminus \mathit{FAV}(\Gamma) \end{aligned}$$

□

Definition 2.3.10. (Instantiation) Let $\sigma = \forall \vec{\alpha}\,\vec{\beta}.C \Rightarrow \hat{\tau}$ be a type scheme. An *instantiation* of σ is a type τ' and a constraint set C' where type variables are substituted by types, and annotation variables are substituted by fresh annotation variables. As usual, this substitution respects bound variables and is capture-avoiding. We write $\sigma \succ C', \hat{\tau}'$ if $C', \hat{\tau}'$ is an instantiation of σ. □

Example 2.3.11. Some type schemes and instantiations.

$$\begin{aligned} &\hat{\tau} \succ \emptyset, \hat{\tau} \\ &\forall \alpha_1 \alpha_2.\{\ \alpha_1 <: \alpha_2\ \} \Rightarrow \alpha_1 \to \alpha_2 \succ \{\ \alpha_3 <: \alpha_4\ \}, \alpha_3 \to \alpha_4 \\ &\forall \alpha_1 \alpha_2.\{\ \alpha_1 <: \alpha_2\ \} \Rightarrow \alpha_1 \to \alpha_2 \succ \{\ \mathit{int} <: \mathit{int}\ \}, \mathit{int} \to \mathit{int} \end{aligned}$$

□

The typing rules for the annotated type system are given in fig. 2.5. Judgements in the type system are of the form $C; \Gamma \vdash e : \hat{\tau}$ where C is a constraint set, Γ a typing environment,

e an expression, and $\hat{\tau}$ an annotated type. Γ assigns types to free program variables in e. C relates free annotation variables and type variables in $\hat{\tau}$.

In some rules we require a fresh annotation variable β. Fresh here means that β must not occur in the free annotation variables of the environment: $\beta \notin \mathrm{FAV}(\Gamma)$. The variable β may however occur in other branches of the type derivation.

[t-const] This rule scheme gives the corresponding types to pure constants, that is *bool* to Boolean constants, *int* to integer constants, and () to ().

[t-var] This rule looks up the type scheme of a variable and instantiates it.

[t-op] This is a rule scheme for all pure operators in the language and typechecks them in the usual way. For example, addition takes two integers and returns an integer, comparisons take two integers and return a Boolean, and so on.

[t-fn] Typechecks abstractions in the usual way. The bound variable is put into the environment and the function body is analysed in this extended environment.

[t-fix] Analyses recursive functions similarly to **[t-fn]**. The function body is analysed under the assumption that f is a function that takes an argument of the type of x.

[t-app] The rule for function application requires that the type of the actual parameter must be a subtype of the type of the formal parameter. The idea is that a function that expects an expensive argument has enough resources to also deal with a cheaper argument. Remember, for base types subtyping just means type equality.

[t-if] Requires the condition to be Boolean, as usual. The overall type of the conditional must be a supertype of the types of both branches. The idea is that if one branch is more expensive than the other, and the context in which the conditional is used has enough resources for the expensive one, it can also deal with the cheaper one.

[t-let] Implements polymorphism and polyvariance. It analyses the defining expression e_x, generalizes its type, and analyses the body under the assumption that x has the generalized type. **[t-let]** plays together with **[t-var]** to allow every use of x to have a different type, as far as instantiation allows.

[t-return] An expression of the form **return** e is a task that can have any cost.

[t-use] Basic tasks have the cost that is specified in the program.

[t-bind] Two tasks, executed in sequence, have the combined cost of the left and the right cost, where reusables can be reused.

[t-para] The parallel composition of two tasks costs as much as the sum of the two tasks individually. No sharing of reusables takes place.

[t-pp] This rule states how to calculate the cost of two parallel tasks e_1 and e_2, whose execution to the current form already costed γ_1 and γ_2 respectively. β_1 and β_2 are the predicted costs of e_1 and e_2. The overall predicted cost of the process pool is the sum of the combinations of the actual costs so far and the predicted rest.

Terminology. Given a program e, an analysis result $C; \Gamma \vdash e : \mathit{task}\ \beta\ \hat{\tau}$, and the least solution s of C, we call $s(\beta)$ the *predicted cost* of e. If $\langle e, \bot \rangle \twoheadrightarrow^* \langle v, \gamma \rangle$, we call γ the *actual cost* of this reduction. We write $s \models C$ for the least solution s of C.

Conjecture 2.3.12. *(Correctness) The analysis is sound with respect to the operational semantics. In other words, the predicted cost of a program is always above the actual cost*

$$\textbf{[t-const]}\quad C; \Gamma \vdash c : \tau_c \qquad \textbf{[t-var]}\ \frac{\Gamma(x) \succ D, \hat{\tau} \qquad C \Vdash D}{C; \Gamma \vdash x : \hat{\tau}}$$

$$\textbf{[t-op]}\ \frac{C; \Gamma \vdash e_1 : \tau^1_\odot \qquad C; \Gamma \vdash e_2 : \tau^2_\odot}{C; \Gamma \vdash e_1 \odot e_2 : \tau_\odot} \qquad \textbf{[t-fn]}\ \frac{C; \Gamma[x \mapsto \hat{\tau}_x] \vdash e_b : \hat{\tau}_b}{C; \Gamma \vdash \textbf{fn}\, x.e_b : \hat{\tau}_x \to \hat{\tau}_b}$$

$$\textbf{[t-fix]}\ \frac{C; \Gamma[f \mapsto \hat{\tau}_x \to \hat{\tau}_b][x \mapsto \hat{\tau}_x] \vdash e_b : \hat{\tau}_b}{C; \Gamma \vdash \textbf{fix} f x.e_b : \hat{\tau}_x \to \hat{\tau}_b}$$

$$\textbf{[t-app]}\ \frac{C; \Gamma \vdash e_1 : \hat{\tau}_1 \to \hat{\tau}_2 \qquad C; \Gamma \vdash e_2 : \hat{\tau}_3 \qquad C \Vdash \hat{\tau}_3 <: \hat{\tau}_1}{C; \Gamma \vdash e_1 e_2 : \hat{\tau}_2}$$

$$\textbf{[t-if]}\ \frac{\begin{array}{lll} & C; \Gamma \vdash e_t : \hat{\tau}_t & C \Vdash \hat{\tau}_t <: \hat{\tau} \\ C; \Gamma \vdash e_c : \mathit{bool} & C; \Gamma \vdash e_e : \hat{\tau}_e & C \Vdash \hat{\tau}_e <: \hat{\tau} \end{array}}{C; \Gamma \vdash \textbf{if}\ e_c\ \textbf{then}\ e_t\ \textbf{else}\ e_e : \hat{\tau}}$$

$$\textbf{[t-let]}\ \frac{\begin{array}{l} C'; \Gamma \vdash e_x : \hat{\tau}_x \\ C; \Gamma[x \mapsto \mathit{generalize}(\hat{\tau}_x, C', \Gamma)] \vdash e_b : \hat{\tau} \end{array}}{C; \Gamma \vdash \textbf{let}\ x = e_x\ \textbf{in}\ e_b : \hat{\tau}}$$

$$\textbf{[t-return]}\ \frac{C; \Gamma \vdash e : \hat{\tau} \qquad C \Vdash \beta \sqsupseteq \bot \qquad \beta \text{ fresh}}{C; \Gamma \vdash \textbf{return}\ e : \mathit{task}\ \beta\ \hat{\tau}}$$

$$\textbf{[t-use]}\ \frac{C; \Gamma \vdash e : \hat{\tau} \qquad C \Vdash \beta \sqsupseteq k \qquad \beta \text{ fresh}}{C; \Gamma \vdash \textbf{use}\ [k]\ e : \mathit{task}\ \beta\ \hat{\tau}}$$

$$\textbf{[t-bind]}\ \frac{\begin{array}{l} C; \Gamma \vdash e_1 : \mathit{task}\ \beta_1\ \hat{\tau}_1 \\ C; \Gamma \vdash e_2 : \hat{\tau}_1 \to \mathit{task}\ \beta_2\ \hat{\tau}_2 \\ C \Vdash \beta \sqsupseteq \beta_1 \boxplus \beta_2 \\ \beta \text{ fresh} \end{array}}{C; \Gamma \vdash e_1 \gg\!= e_2 : \mathit{task}\ \beta\ \hat{\tau}_2} \qquad \textbf{[t-para]}\ \frac{\begin{array}{l} C; \Gamma \vdash e_1 : \mathit{task}\ \beta_1\ () \\ C; \Gamma \vdash e_2 : \mathit{task}\ \beta_2\ () \\ C \Vdash \beta \sqsupseteq \beta_1 + \beta_2 \\ \beta \text{ fresh} \end{array}}{C; \Gamma \vdash e_1\ \&\ e_2 : \mathit{task}\ \beta\ ()}$$

$$\textbf{[t-pp]}\ \frac{\begin{array}{l} C; \Gamma \vdash e_1 : \mathit{task}\ \beta_1\ () \\ C; \Gamma \vdash e_2 : \mathit{task}\ \beta_2\ () \\ C \Vdash \beta \sqsupseteq (\beta_1 \boxplus \gamma_1) + (\beta_2 \boxplus \gamma_2) \\ \beta \text{ fresh} \end{array}}{C; \Gamma \vdash \mathit{pp}(e_1\ \&\ e_2, \gamma_1, \gamma_2) : \mathit{task}\ \beta\ ()}$$

Figure 2.5: The annotated type system

of any possible reduction. Formally,

$$\begin{array}{rrcl} \textit{assume} & C;\Gamma & \vdash & e : \textit{task}\ \beta\ \hat{\tau} \\ \textit{and} & \langle e, \bot\rangle & \twoheadrightarrow^* & \langle e', \gamma'\rangle \\ \textit{and} & C';\Gamma & \vdash & e' : \textit{task}\ \beta'\ \hat{\tau}. \\ \textit{Let} & s & \models & C \\ \textit{and} & s' & \models & C'. \\ \textit{Then} & s(\beta) & \sqsupseteq & s'(\beta') \boxplus \gamma'. \end{array}$$

□

2.4 Implementation

This section describes an algorithm that implements the type system. We implemented the algorithm in Clean, but this chapter describes it in an abstract way that ignores many implementation details. The source code, together with example programs and unit tests can be found online [Klinik, 2017]. The implementation is intended to be a proof-of-concept. No effort has been put into optimization.

The analysis works in three steps. First, there is a modified version of algorithm $\mathscr{W}$ that infers types and collects constraints about type and annotation variables. Second, the subsumption constraint solver decomposes subtype constraints into effect constraints and unifications. Third, and only if the top-level expression is of type *task*, a worklist algorithm calculates a solution of the effect constraints. The cost of the analysed program is the entry in the solution that corresponds to the annotation variable of the top-level expression. If the top-level expression is not a task, and therefore does not have a definite cost, the analysis just reports the type and the constraints.

Our system deals with two kinds of variables: annotation variables and type variables. Substitutions θ replace type variables by types and annotation variables by annotation *variables*. We take the liberty to combine type and annotation substitutions for the sake of brevity. Type and annotation substitutions are denoted by $[\alpha \mapsto \hat{\tau}]$ and $[\beta_1 \mapsto \beta_2]$ respectively. The empty substitution is denoted by $[\,]$. Substitutions are applicable to types, type schemes and constraints, and extend pointwise to environments and constraint sets. When applied to type schemes, substitutions respect bound variables. Substitutions can be composed, which is denoted by $\theta_2 \circ \theta_1$, and defined as $(\theta_2 \circ \theta_1)\hat{\tau} = \theta_2(\theta_1\hat{\tau})$.

2.4.1 Algorithm W and Unification

Type inference uses a unification algorithm. Unification takes two types and returns a substitution if the types can be unified, and an error otherwise. Figure 2.6 shows the unification algorithm $\mathscr{U}$. The difference between unification in textbook Hindley-Milner algorithms and our algorithm is that ours has to deal with task types, which carry annotation variables. As such, unification is also applicable to annotation variables. The relevant clauses are in lines (2.1) and (2.2) in fig. 2.6. In line (2.1), tasks are unified by unifying their annotation variables and their return types. In line (2.2), annotation variables are unified by generating a substitution.

The first step of the analysis is a variant of algorithm $\mathscr{W}$. It takes as input a program e in our language and an environment Γ, and returns a triple of an annotated type $\hat{\tau}$, a

$$
\begin{aligned}
&\mathcal{U}(bool, bool) = [\,] \\
&\mathcal{U}(int, int) = [\,] \\
&\mathcal{U}(\alpha, \alpha) = [\,] \\
&\mathcal{U}(\alpha, \hat{\tau}) = [\alpha \mapsto \hat{\tau}] \text{ if } \alpha \notin FTV(\hat{\tau}), \text{Error otherwise.} \\
&\mathcal{U}(\hat{\tau}, \alpha) = \mathcal{U}(\alpha, \hat{\tau}) \\
&\mathcal{U}(\hat{\tau}_1 \to \hat{\tau}_2, \hat{\tau}_3 \to \hat{\tau}_4) = \theta_2 \circ \theta_1 \text{ where} \\
&\quad \theta_1 = \mathcal{U}(\hat{\tau}_1, \hat{\tau}_3) \\
&\quad \theta_2 = \mathcal{U}(\theta_1 \hat{\tau}_2, \theta_1 \hat{\tau}_4) \\
&\mathcal{U}(task\ \beta_1\ \hat{\tau}_1, task\ \beta_2\ \hat{\tau}_2) = \theta_2 \circ \theta_1 \text{ where} \qquad (2.1) \\
&\quad \theta_1 = \mathcal{U}(\beta_1, \beta_2) \\
&\quad \theta_2 = \mathcal{U}(\theta_1 \hat{\tau}_1, \theta_1 \hat{\tau}_2) \\
&\mathcal{U}(\hat{\tau}_1, \hat{\tau}_2) = \text{Error, cannot unify types.} \\
&\mathcal{U}(\beta, \beta) = [\,] \\
&\mathcal{U}(\beta_1, \beta_2) = [\beta_1 \mapsto \beta_2] \qquad (2.2)
\end{aligned}
$$

Figure 2.6: The unification algorithm

substitution θ, and a set of constraints C. The returned results are such that if s is a solution of θC, and e is of type *task*, that is $\theta\hat{\tau} = task\ \beta\ \hat{\tau}_1$, then $s(\beta)$ is an upper bound of the cost of e. Algorithm $\mathcal{W}$ is given in fig. 2.7.

Figure 2.7 shows type inference for pure expressions.

(2.1), (2.2) These clauses typecheck Boolean and integer constants. Such expressions are of type *bool* and *int* respectively.

(2.3) The clause for variables looks up the type scheme of x in the environment and instantiates it. Instantiation means that fresh type- and annotation variables are generated and substituted for all the bound ones in the type and the constraint set.

(2.4) The clause for abstractions puts the bound variable x into the environment with a fresh type variable α and typechecks the function body.

(2.5) Recursive functions are typechecked by putting the function and the argument with fresh type variables into the environment and then checking the function body. Then, the constraints from the function body are solved. This extracts as much information as possible at that point out of the constraint set. In particular, it might generate effect constraints from subsumption constraints. This is needed because the subsequent widening needs to see all effect constraints, or else it might miss a widening opportunity, leading to an infinite loop in the solver after typechecking. The result of constraint solving is a set of effect constraints C_e, a set of unresolved subsumption constraints C_s, and a substitution. The resulting constraints are widened using the function *widen*.

Definition 2.4.1. (Widening) The function *widen* takes a set of constraints and yields a set of constraints where the variable in each constraint is widened with its own

right-hand side. Subsumption constraints are left untouched. Formally:

$$\begin{aligned} \mathit{widen}(C) = &\{\, \beta \sqsupseteq \beta \nabla \varphi \mid \beta \sqsupseteq \varphi \in C \,\} \\ &\cup \{\, \hat{\tau}_1 <: \hat{\tau}_2 \mid \hat{\tau}_1 <: \hat{\tau}_2 \in C \,\} \end{aligned}$$ □

Finally, unification of the type of the body and the return type of the function makes sure that all acquired information is contained in the result. This clause is the only place where widening is applied, because it is the main source of recursive constraints. Unfortunately there are other situations in which recursive constraints can arise, as discussed in example 2.5.10.

(2.6) The clause for applications typechecks function and argument expressions independently and uses unification to make sure that the function expression has function type. In contrast to textbook Hindley-Milner, the clause does not unify the types of the formal and actual arguments. Instead it generates a subtyping constraint that requires the actual argument to be a subtype of the formal argument.

(2.7) The clause for conditionals typechecks the condition and uses unification to make sure that it is of type *bool*. It then typechecks the then- end else-branches independently and generates two subtyping constraints which make sure that the type of the conditional is a supertype of both branches.

(2.8) The clause for let-bindings first typechecks the defining expression of x and then generalizes the type and constraints resulting from that. The body of the let-binding is then typechecked in an environment where x maps to its type scheme.

(2.9) The clause for pure operators does not deal with annotations at all. It typechecks both arguments and uses unification to make sure that the actual argument types match the formal argument types of the operator.

(2.10) This clause handles basic tasks. It generates a constraint that makes sure that the constraint solver takes the cost of the task into account.

(2.11) The clause for sequential task composition looks complex but holds no surprises. The left argument must be a task and the right argument a function that accepts the task's value. The function must yield a task. The clause generates a constraint that ensures that the cost of the overall expression is the combination of the cost of the left and right arguments.

(2.12) Parallel task composition is simpler than sequential composition. Both arguments must be tasks that return the unit type. The value of the overall expression is also unit. See section 2.2.1 for a rationale. The cost of the overall expression is the sum of the costs of the arguments.

2.4.2 Subsumption Constraint Solving

This section motivates and describes the subsumption constraint solver. In many type and effect systems, all types are annotated with effects. An example is exception analysis, where expressions of any type can throw exceptions. In our case however, only tasks have annotations because only tasks have costs.

Our algorithm $\mathscr{W}$ performs resource analysis by traversing the abstract syntax tree. This always happens in a particular order, and as with Hindley-Milner type inference we face the problem that information about the type of some expressions becomes available

$$\mathscr{W}(\Gamma, \textbf{True}) = \mathscr{W}(\Gamma, \textbf{False}) = \langle bool,\ [\,],\ \emptyset \rangle \tag{2.1}$$

$$\mathscr{W}(\Gamma, n) = \langle int,\ [\,],\ \emptyset \rangle \tag{2.2}$$

$$\mathscr{W}(\Gamma, x) = \langle \hat{\tau},\ [\,],\ C \rangle \text{ where} \tag{2.3}$$
$$\langle \hat{\tau}, C \rangle = inst(\Gamma(x))$$

$$\mathscr{W}(\Gamma, \textbf{fn}\, x.e_b) = \langle \theta_b \alpha \to \hat{\tau}_b,\ \theta_b,\ C_b \rangle \text{ where} \tag{2.4}$$
$$\alpha \text{ fresh}$$
$$\langle \hat{\tau}_b, \theta_b, C_b \rangle = \mathscr{W}(\Gamma[x \mapsto \alpha], e_b)$$

$$\mathscr{W}(\Gamma, \textbf{fix} f x.e_b) = \langle \theta_{123}\alpha_x \to (\theta_3 \circ \theta_2)\hat{\tau}_r,\ \theta_{123},\ C_2 \cup C_s \rangle \tag{2.5}$$
where α_x, α_r fresh
$$\langle \hat{\tau}_r, \theta_1, C_1 \rangle = \mathscr{W}(\Gamma[f \mapsto \alpha_x \to \alpha_r][x \mapsto \alpha_x], e_b)$$
$$\theta_2 = \mathscr{U}(\hat{\tau}_r, \theta_1 \alpha_r)$$
$$\langle C_e, C_s, \theta_3 \rangle = solveSubsumptions(\theta_2 C_1)$$
$$C_2 = widen(\theta_2 C_e)$$
$$\theta_{123} = \theta_3 \circ \theta_2 \circ \theta_1$$

$$\mathscr{W}(\Gamma, e_1 e_2) = \langle \theta_3 \alpha_2,\ \theta_3 \circ \theta_2 \circ \theta_1, \tag{2.6}$$
$$(\theta_3 \circ \theta_2)C_1 \cup\ \theta_3 C_2\ \cup\ \{\, \theta_3 \hat{\tau}_2 <: \theta_3 \alpha_1 \,\} \rangle \text{ where}$$
$$\alpha_1, \alpha_2 \text{ fresh}$$
$$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathscr{W}(\Gamma, e_1)$$
$$\langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathscr{W}(\theta_1 \Gamma, e_2)$$
$$\theta_3 = \mathscr{U}(\theta_2 \hat{\tau}_1,\ \alpha_1 \to \alpha_2)$$

$$\mathscr{W}(\Gamma, \textbf{if}\ e_c\ \textbf{then}\ e_t\ \textbf{else}\ e_e) = \langle \theta_4 \alpha,\ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, \tag{2.7}$$
$$(\theta_4 \circ \theta_3 \circ \theta_2)C_1 \cup (\theta_4 \circ \theta_3)C_2 \cup \theta_4 C_3$$
$$\cup\ \{\, (\theta_4 \circ \theta_3)\hat{\tau}_2 <: \alpha,\ \theta_4 \hat{\tau}_3 <: \alpha \,\} \rangle \text{ where}$$
$$\alpha \text{ fresh}$$
$$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathscr{W}(\Gamma, e_c)$$
$$\langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathscr{W}(\theta_1 \Gamma, e_t)$$
$$\langle \hat{\tau}_3, \theta_3, C_3 \rangle = \mathscr{W}((\theta_2 \circ \theta_1)\Gamma, e_e)$$
$$\theta_4 = \mathscr{U}((\theta_3 \circ \theta_2)\hat{\tau}_1, bool)$$

$$\mathscr{W}(\Gamma, \textbf{let}\ x = e_x\ \textbf{in}\ e_b) = \langle \hat{\tau}_2,\ \theta_2 \circ \theta_1,\ \theta_2 C_g \cup C_2 \rangle \tag{2.8}$$
where
$$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathscr{W}(\Gamma, e_x)$$
$$\Gamma' = \theta_1 \Gamma$$
$$\langle \sigma_1, C_g \rangle = generalize(\hat{\tau}_1, \Gamma', C_1)$$
$$\langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathscr{W}(\Gamma'[x \mapsto \sigma_1], e_b)$$

Figure 2.7: Algorithm $\mathscr{W}$

$$\mathscr{W}(\Gamma, e_1 \odot e_2) = \langle \tau_\odot,\ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, \tag{2.9}$$
$(\theta_4 \circ \theta_3 \circ \theta_2)C_1 \cup (\theta_4 \circ \theta_3)C_2 \rangle$ where
$\langle \tau_1, \theta_1, C_1 \rangle = \mathscr{W}(\Gamma, e_1)$
$\langle \tau_2, \theta_2, C_2 \rangle = \mathscr{W}(\theta_1\Gamma, e_2)$
$\theta_3 = \mathscr{U}(\theta_2\tau_1, \tau^1_\odot)$
$\theta_4 = \mathscr{U}(\theta_3\tau_2, \tau^2_\odot)$
such that $\tau_\odot, \tau^1_\odot, \tau^2_\odot$ match the respective operator:
$\tau_+ = \tau^1_+ = \tau^2_+ = \mathit{int}$
$\tau_< = \mathit{bool}, \tau^1_< = \tau^2_< = \mathit{int}$
and similar for the other binary operators …

$$\mathscr{W}(\Gamma, \textbf{use}\ [k]\ e_1) = \langle \mathit{task}\ \beta\ \hat{\tau}_1, \theta_1, \{\ \beta \sqsupseteq k\ \} \cup C_1 \rangle \tag{2.10}$$
where β fresh
$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathscr{W}(\Gamma, e_1)$

$$\mathscr{W}(\Gamma, e_1 \gg\!= e_2) = \langle \mathit{task}\ \beta\ ((\theta_4 \circ \theta_3)\alpha_2), \tag{2.11}$$
$\theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1,$
$(\theta_4 \circ \theta_3 \circ \theta_2)C_1 \cup (\theta_4 \circ \theta_3)C_2$
$\quad \cup \{\ \beta \sqsupseteq ((\theta_4 \circ \theta_3)\beta_1) \boxplus (\theta_4\beta_2)\ \} \rangle$
where
$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathscr{W}(\Gamma, e_1)$
$\langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathscr{W}(\theta_1\Gamma, e_2)$
$\beta, \beta_1, \beta_2, \alpha_1, \alpha_2$ fresh
$\theta_3 = \mathscr{U}(\theta_2\hat{\tau}_1,\ \mathit{task}\ \beta_1\ \alpha_1)$
$\theta_4 = \mathscr{U}(\theta_3\hat{\tau}_2,\ (\theta_3\alpha_1) \to \mathit{task}\ \beta_2\ \alpha_2)$

$$\mathscr{W}(\Gamma, e_1\ \&\ e_2) = \langle \mathit{task}\ \beta\ (), \tag{2.12}$$
$\theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1,$
$(\theta_4 \circ \theta_3 \circ \theta_2)C_1 \cup (\theta_4 \circ \theta_3)C_2$
$\quad \cup \{\ \beta \sqsupseteq ((\theta_4 \circ \theta_3)\beta_1) + (\theta_4\beta_2)\ \} \rangle$
where
$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathscr{W}(\Gamma, e_1)$
$\langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathscr{W}(\theta_1\Gamma, e_2)$
β, β_1, β_2 fresh
$\theta_3 = \mathscr{U}(\theta_2\hat{\tau}_1, \mathit{task}\ \beta_1\ ())$
$\theta_4 = \mathscr{U}(\theta_3\hat{\tau}_2, \mathit{task}\ \beta_2\ ())$

Figure 2.7: Algorithm $\mathscr{W}$ *(cont.)*

$$solveSubsumption(\beta \sqsupseteq \varphi) = \langle \{\, \beta \sqsupseteq \varphi \,\},\ \emptyset,\ \emptyset,\ [\,] \rangle \tag{2.1}$$

$$solveSubsumption(\alpha_1 <: \alpha_2) = \langle \emptyset,\ \{\, \alpha_1 <: \alpha_2 \,\},\ \emptyset,\ [\,] \rangle \tag{2.2}$$

$$solveSubsumption(task\ \beta_1\ \hat{\tau}_1 <: task\ \beta_2\ \hat{\tau}_2) = \tag{2.3}$$
$$\langle \{\, \beta_2 \sqsupseteq \beta_1 \,\},\ \emptyset,\ \{\, \hat{\tau}_1 <: \hat{\tau}_2 \,\},\ [\,] \rangle$$

$$solveSubsumption(\hat{\tau}_1 \to \hat{\tau}_2 <: \hat{\tau}_3 \to \hat{\tau}_4) = \tag{2.4}$$
$$\langle \emptyset,\ \emptyset,\ \{\, \hat{\tau}_3 <: \hat{\tau}_1, \hat{\tau}_2 <: \hat{\tau}_4 \,\},\ [\,] \rangle$$

$$solveSubsumption(task\ \beta_1\ \hat{\tau}_1 <: \alpha_1) = \tag{2.5}$$
$$\langle \emptyset,\ \emptyset,\ \{\, task\ \beta_1\ \hat{\tau}_1 <: \theta_1\alpha_1 \,\},\ \theta_1 \rangle \text{ where}$$
$$\alpha_2, \beta_2 \text{ fresh}$$
$$\theta_1 = \mathcal{U}(\alpha_1, task\ \beta_2\ \alpha_2)$$

$$solveSubsumption(\alpha_1 <: task\ \beta_1\ \hat{\tau}_1) = \tag{2.6}$$
$$\langle \emptyset,\ \emptyset,\ \{\, \theta_1\alpha_1 <: task\ \beta_1\ \hat{\tau}_1 \,\},\ \theta_1 \rangle \text{ where}$$
$$\alpha_2, \beta_2 \text{ fresh}$$
$$\theta_1 = \mathcal{U}(\alpha_1, task\ \beta_2\ \alpha_2)$$

$$solveSubsumption(\hat{\tau}_1 \to \hat{\tau}_2 <: \alpha_1) = \tag{2.7}$$
$$\langle \emptyset,\ \emptyset,\ \{\, \hat{\tau}_1 \to \hat{\tau}_2 <: \theta_1\alpha_1 \,\},\ \theta_1 \rangle \text{ where}$$
$$\alpha_2, \alpha_3 \text{ fresh}$$
$$\theta_1 = \mathcal{U}(\alpha_1, \alpha_2 \to \alpha_3)$$

$$solveSubsumption(\alpha_1 <: \hat{\tau}_1 \to \hat{\tau}_2) = \tag{2.8}$$
$$\langle \emptyset,\ \emptyset,\ \{\, \theta_1\alpha_1 <: \hat{\tau}_1 \to \hat{\tau}_2 \,\},\ \theta_1 \rangle \text{ where}$$
$$\alpha_2, \alpha_3 \text{ fresh}$$
$$\theta_1 = \mathcal{U}(\alpha_1, \alpha_2 \to \alpha_3)$$

$$solveSubsumption(\hat{\tau}_1 <: \hat{\tau}_2) = \langle \emptyset,\ \emptyset,\ \emptyset,\ \mathcal{U}(\hat{\tau}_1, \hat{\tau}_2) \rangle \tag{2.9}$$

Figure 2.8: The subsumption constraint solver

only when the algorithm advances further into the AST. Hindley-Milner solves this by generating and solving type equality constraints. Our system features subtyping, which cannot be solved by unification. Instead of type equality constraints, we must generate subeffect constraints, but only if the involved types are tasks. For base types we want regular unification. If the algorithm cannot immediately determine whether to use unification or subeffecting, it has to defer the decision until more information about the types is available.

Generating subsumption constraints, which are later resolved by either unification or turning them into subeffect constraints, is conceptually similar to annotating all types and dropping the annotation when it becomes clear that a type is not a task. We chose for the former because it allows experimenting with different forms of subtyping and subeffecting just by modifying the constraint solver. The constraint solver presented in this chapter implements full subtyping.

Subsumption constraint solving happens after algorithm $\mathscr{W}$, and its goal is to decompose all subsumption constraints as far as possible to extract effect constraints according to the subtyping rules. Subsumption solving is a many-pass procedure over the list of constraints that algorithm $\mathscr{W}$ collects. Unification takes place in every pass, from which the solver might learn more about the type variables in the constraint set. As long as the solver learns more details, it needs to continue performing passes. The procedure eventually terminates because the types involved in subsumption constraints become structurally smaller in each pass, and our system does not have recursive types.

2.4.2.1 Solving Constraint Sets

The function *solveSubsumption* (fig. 2.8) handles a single constraint. To solve a constraint set, we must perform multiple passes over the whole set. The function *solveSubsumptions* (not shown) calls *solveSubsumption* on every constraint, and performs as many passes as necessary. The function *solveSubsumption* gets a single constraint as input and returns a tuple consisting of effect constraints, unresolved subsumption constraints, freshly generated subsumption constraints, and a substitution. By looking at these, *solveSubsumptions* determines if another pass is needed. The effect constraints are the actual output of the solving process. They are collected and later passed on to the effect constraint solver. Unresolved constraints are constraints about which the current pass does not yet have enough information. The solver must keep track of them, because as subsequent passes solve more constraints, and learns more about the type variables involved, solving them might become possible. Freshly generated subsumptions have been generated in the current pass. If there are any of those, another pass is required. If a pass uses unification to solve a constraint, a substitution is generated. The presence of a substitution indicates that a pass learned more about some type variables. Whenever a pass returns a substitution, the substitution must be applied to all unresolved constraints and another pass is required.

2.4.2.2 Solving Individual Constraints

The function *solveSubsumption* in fig. 2.8 handles constraints as follows.

(2.1) This clause just filters out effect constraints. It is needed because we mix effect and subsumption constraints in the same set.

(2.2) This clause handles constraints involving two type variables. Nothing is known about them at this point, so they go immediately to the set of unresolved constraints.
(2.3) This clause handles constraints involving two task types, according to the subtyping rules. Subtyping for tasks $\mathit{task}\ \beta_1\ \hat{\tau}_1 <: \mathit{task}\ \beta_2\ \hat{\tau}_2$ requires that the cost for the right task is above the cost of the left task: $\beta_2 \sqsupseteq \beta_1$, and that the types of the task values are again in the subtyping relation $\hat{\tau}_1 <: \hat{\tau}_2$.
(2.4) This clause handles constraints involving two function types. It implements co- and contravariance of function types.
(2.5), (2.6), (2.7), (2.8) These clauses handle constraints where one of the types is a type variable and the other a type constructor. In all of these cases the solver learns that a variable must be a task or a function, respectively, and records this fact by generating a new constraint involving some fresh type variables.
(2.9) This is the catch-all clause. It handles all other constraints, in particular those involving base types, and those that cause type errors.

2.4.3 Effect Constraint Solving

The effect constraint solver is a translation to functional code of the worklist algorithm found in [Nielson et al., 1999, chap. 6]. We use set notation for the worklist for brevity. The solver takes as input the set of effect constraints generated by the analysis so far. The output is a mapping from annotation variables to costs that satisfies the input constraints. Such a mapping is called a *solution*. The function *solve*, which implements the effect constraint solver, is given in fig. 2.9.

(2.1) Starts the iteration with initial parameters. The initial worklist is the set of constraints, which means every constraint is examined at least once.
(2.2) The initial solution maps every annotation variable to bottom.
(2.3) The mapping *influences* stores for every annotation variable β the set of constraints where β occurs on the right-hand side, which is the set of all constraints that β influences.
(2.4) The base case of the iteration. If the worklist is empty, the algorithm terminates.
(2.5) The worklist consists of at least one constraint $\beta \sqsupseteq \varphi$ and some rest. The new cost of β is calculated in line **(2.6)** by taking the least upper bound of the cost so-far $s(\beta)$ and the evaluation of the annotation φ under s. The lub with $s(\beta)$ is needed because there may be several constraints with β on the left-hand side, and this way the solver keeps track of the most expensive one. The flag *dirty* indicates whether β has gotten an increased cost, in which case all constraints that depend on β must be re-evaluated.
(2.7) If necessary, the worklist for the next iteration is extended by all the constraints that β influences. The solution for the next iteration is updated with the new cost for β.

2.5 Discussion

In this section we look at example programs, discuss which challenges they pose to the analysis and explain why our method can or cannot deal with them. The employment of techniques such as subtyping and polyvariance to reduce poisoning is state-of-the-art in

$$solve(C) = iterate(influences,\ C,\ initSolution) \tag{2.1}$$
where
$$initSolution = [\ \beta \mapsto \bot \mid \beta \in FAV(C)\ ] \tag{2.2}$$
$$influences = [\ \beta \mapsto infl'(\beta) \mid \beta \in FAV(C)\ ] \tag{2.3}$$
$$infl'(\beta) = \{\ \beta_1 \sqsupseteq \varphi \mid \beta_1 \sqsupseteq \varphi \in C, \beta \in FAV(\varphi)\ \}$$
$$iterate(\_,\ \emptyset,\ s) = s \tag{2.4}$$
$$iterate(influences,\ \{\ \beta \sqsupseteq \varphi\ \} \cup rest,\ s) = \tag{2.5}$$
$$iterate(influences,\ worklist',\ s') \text{ where}$$
$$new\beta = [\![\varphi]\!]s \sqcup s(\beta) \tag{2.6}$$
$$dirty = new\beta \sqsupset s(\beta)$$
$$worklist' = \begin{cases} rest \cup influences(\beta) & \text{if } dirty \\ rest & \text{otherwise} \end{cases} \tag{2.7}$$
$$s' = s[\beta \mapsto new\beta]$$

Figure 2.9: The effect constraint solver

program analysis, see for example [Gedell et al., 2006]. In this section we explain how our system makes use of these techniques to increase the precision of the analysis.

In the following examples we talk about programs of type *task*. To express the cost of a task, the analysis generates a type with annotation variable like *task* β *int* and some constraint like $\beta \sqsupseteq [3C]$. Together these indicate that the task costs $3C$ to execute. When discussing example programs it is cumbersome to explicitly mention constraints. We take the liberty to put costs directly in task types and write *task* $[3C]$ *int*.

2.5.1 Good Examples

Let us first look at some example programs that our analysis can deal with nicely. The subsequently described features all solve different kinds of problems, but often their uses overlap. This means there are programs that can be analysed precisely by more than one of them. Nonetheless, there are corner cases that can only be solved by one of them.

Subtyping. There are two precursors to proper subtyping. A simple form of subtyping, called *creation-site subeffecting*, is necessary for type and effect systems to be conservative extensions of their underlying type systems. Being a conservative extension means that all programs typeable in the underlying type system can be analysed. Creation-site subeffecting in our system means that tasks with cost k can always be considered to cost more. The prime example where this is necessary is the typing rule for conditionals. The underlying type system requires the types of the branches to be identical, which means that their costs must be identical. Creation-site subeffecting allows the cost of the cheaper task to be increased to match the cost of the more expensive one.

Creation-site subeffecting while sound, however, leads to poor results in certain situations. This effect is called *poisoning*, and happens when the same task t is used in different

contexts. The cost of t is increased to match the requirements of the most expensive context, raising its cost in the other contexts as well, where it could be taken as cheaper.

Example 2.5.1. Consider the following program in a system without polymorphism but with creation-site subeffecting.

$$\begin{aligned}&\textbf{let } t = \textbf{use } [1C]\ 0 \textbf{ in}\\&\textbf{let } s = \textbf{use } [3C]\ 0 \textbf{ in}\\&\textbf{let } u = \textbf{if True then } s \textbf{ else } t \textbf{ in } t\end{aligned}$$

The conditional in the unused u forces the costs of the defining expressions of s and t to be identical, $3C$. The overall expression evaluates to just t, which actually costs $1C$ but is analysed as costing $3C$. □

To alleviate this problem, one can allow the costs of task expressions to be adjusted not at the places where they are defined, but where they are used. This technique is called *use-site subeffecting*, and allows the cost of a task to be adjusted in each context individually. In the above example this would mean the cost of t only increases in its use in the conditional, while the use in the main expression is unaffected. Use-site subeffecting is achieved in a type system by moving the subeffect conditions from the axioms for constants to all the rules where unification happens.

Subeffecting only applies to the annotation of the top-level type constructor of a type. It does not apply to annotations deeper in compound types. In our case such annotations occur in functions that have tasks as arguments or return values, or tasks that return tasks as values. This is where subtyping comes in. Subtyping allows subeffecting at any place in a compound type, which is important for lambda-bound higher-order functions and higher-order tasks.

Let-polymorphism. Our system features polymorphism in the usual Hindley-Milner style. When the type of a let-bound variable is not fully determined, remaining type variables are generalized. When the variable is used, the type variables get instantiated to match the type required by the context. This way the same identifier can have many types that do not influence each other.

Example 2.5.2. The classical example for polymorphism is the identity function. Consider the following program in our system.

$$\begin{aligned}&\textbf{let } id = \textbf{fn } x.x \textbf{ in}\\&id(\textbf{use } [1C]\ (id\, 0))\end{aligned}$$

The function id has type $\forall\alpha.\alpha \to \alpha$, which can be instantiated differently in the body of the let. The outer occurrence of id gets type $task\,[1C]\,int \to task\,[1C]\,int$ and the inner occurrence $int \to int$. □

Let-polyvariance. Polyvariance is needed when the types of let-bound identifiers are not general enough for polymorphism to apply. Polyvariance is similar to polymorphism but binds annotation variables instead of type variables.

Example 2.5.3. In the following example, the function *twoTimes* runs a given task two times in sequence. In the body of the let it is applied to tasks with different costs.

$$\begin{aligned}&\textbf{let } \mathit{twoTimes} = \textbf{fn}\, x.x \gg x \textbf{ in}\\ &\mathit{twoTimes}\ (\textbf{use}\ [2C]\ 0) \gg \mathit{twoTimes}\ (\textbf{use}\ [1C]\ 0)\end{aligned}$$

To get the expected analysis result, $6C$, *twoTimes* needs the following types in the first and second call respectively.

$$\mathit{task}\,[2C]\,\mathit{int} \to \mathit{task}\,[4C]\,\mathit{int} \tag{2.1}$$
$$\mathit{task}\,[1C]\,\mathit{int} \to \mathit{task}\,[2C]\,\mathit{int} \tag{2.2}$$

Polymorphism does not help because the sequence operator in the definition of *twoTimes* forces its type to be a function from tasks to tasks, hence there is no type variable that can be instantiated to task types with different costs. Without polyvariance the type system must give *twoTimes* the more expensive type (2.1) for both calls, leading to a result of $8C$. With polyvariance the type system can assign the following type to *twoTimes*.

$$\forall \beta.\mathit{task}\ \beta\ \mathit{int} \to \mathit{task}\ (\beta \boxplus \beta)\ \mathit{int}$$

This type can be instantiated to the types (2.1) and (2.2) above. □

Example 2.5.4. In example 2.5.1 we argued that subtyping prevents poisoning. In fact, because the identifiers in that example are let-bound, polyvariance also prevents poisoning. In the following program however, polyvariance does not apply because s and t are lambda-bound instead of let-bound.

$$\begin{aligned}&(\textbf{fn}\ t.\textbf{fn}\ s.\mathit{const}\ t\ (\textbf{if True then}\ s\ \textbf{else}\ t))\\ &\quad(\textbf{use}\ [1C]\ 0)(\textbf{use}\ [3C]\ 0)\end{aligned}$$

Only subtyping can prevent poisoning so that the expression has cost $1C$. □

Recursion. To analyse recursive functions, the algorithm makes use of a technique called *widening*. Widening solves the problem that recursion depth, or termination for that matter, is Turing complete. A recursive function that uses some resource in each iteration could potentially require an infinite amount of resources. A naive implementation using constraints leads to recursive constraints, that is, a group of constraints where an annotation variable occurs on both the left and the right-hand side of constraints. In the simplest case this happens in the same constraint, for example:

$$\{\, \beta \sqsupseteq \beta \boxplus \beta, \beta \sqsupseteq [1C] \,\}.$$

This constraint set has a solution in our domain, namely $[\beta \mapsto \infty]$. Kleene-style fixpoint iteration however cannot compute this solution, because starting at $[\beta \mapsto \bot]$ the iteration diverges. Widening guarantees that fixpoint iteration terminates in the presence of recursive constraints. The widening operator ∇, formally defined in definition 2.2.10, takes two arguments and yields infinity if the right argument is above the left one. Our algorithm

applies widening to all constraints coming from the bodies of recursive functions. See definition 2.4.1. The above constraint set becomes:

$$\{\, \beta \sqsupseteq \beta \nabla \beta \boxplus \beta, \beta \sqsupseteq \beta \nabla [1C] \,\}.$$

This causes the fixpoint iteration to stepwise calculate the following values for β. After the third step, the value for β no longer changes and the process terminates.

$$\begin{aligned}
\beta &= \bot \\
\beta &= 1C \\
\beta &= 1C \nabla (1C \boxplus 1C) = 1C \nabla 2C = \infty C
\end{aligned}$$

Example 2.5.5. The following program is an infinite recursion where each iteration costs one unit of a consumable resource C and one unit of a reusable resource R. The algorithm estimates the cost of the program as expected with $\infty C + 1R$.

$$(\mathbf{fix} f x.x \gg (f x))(\mathbf{use}\ [1C + 1R]\ 0) \qquad \square$$

Example 2.5.6. In the following program, the function doubles its argument in each recursive call. The parameter costs $1R$, but reusables cannot be shared between parallel tasks. The cost of each iteration is twice the cost of the previous one. Our algorithm estimates the cost of the program with ∞R.

$$(\mathbf{fix} f x.x \gg (f (x \,\&\, x)))(\mathbf{use}\ [1R]\ ()) \qquad \square$$

Higher-order tasks. Tasks not only consume resources, they also produce values. These values can be of any type, in particular they can be tasks. This is useful in the world of iTasks, the catchphrase being "managing tasks is a task". Think about a person deciding which task to execute. The task of making the decision itself has a cost, as does the resulting task. Another example could be that the procedure of investigating the nature of a fire has as result the corresponding firefighting task.

Example 2.5.7. In the following program, *decide* is a task whose result is a task. The expression c is some condition whose details are not important.

$$\begin{aligned}
&\mathbf{let}\ \mathit{decide} = \mathbf{use}\ [1C]\ (\mathbf{if}\ c\ \mathbf{then} \\
&\quad \mathbf{use}\ [3R]\ 0\ \mathbf{else}\ \mathbf{use}\ [4R]\ 0)\ \mathbf{in} \\
&\mathbf{let}\ \mathit{execute} = \mathit{id}\ \mathbf{in} \\
&\mathit{decide} \gg\!= \mathit{execute}
\end{aligned}$$

The task *decide* has type $\mathit{task}\,[1C]\,(\mathit{task}\,[4R]\,\mathit{int})$. The overall program has predicted cost $1C + 4R$, because first the decision is made and then the resulting task is executed. $\square$

2.5.2 Challenging Examples

The analysis always gives safe approximations, which means that the cost of actually running a program is never higher than the analysis prediction. In the examples so far the predicted costs match what a programmer would reasonably expect by inspecting the programs. In the following examples we look at programs where the analysis gives results that are unexpected unless the programmer is familiar with the specifics of the analysis algorithm.

Widening. The widening operator yields infinity immediately when its right-hand side is above the left-hand side. This is a quite aggressive strategy which gives bad results for constraint sets where iteration would otherwise stabilize after a finite number of steps.

Example 2.5.8. The following program uses the fixpoint combinator but has no recursive calls. This causes the constraints from the function body to be unnecessarily subjected to widening. Solving the constraints involves an intermediate solution where one of the widened constraints goes to infinity, whereas without widening, iteration would stabilize at a finite value after a couple of rounds.

$$(\mathbf{fix}\, f\, x.\mathbf{if}\ \mathbf{True}\ \mathbf{then}\ x\ \mathbf{else}\ (x \gg x))(\mathbf{use}\ [1C]\ 0)$$

The analysis outcome we would like to have is $2C$ because that is the worst case of the conditional. The prediction our algorithm gives is ∞C, for the reasons described above. □

Lambda-bound functions. Polymorphism and polyvariance only apply to let-bound variables. Lambda-bound variables are still subject to subtyping, which allows good results if they are used as arguments in function calls. If a lambda-bound variable is used in the function position in a function call, neither polymorphism nor subtyping applies, which can result in poisoning.

Example 2.5.9. In the following example, the identity function is lambda-bound to f. There are two applications of f, of which the second is ignored.

$$(\mathbf{fn}\, f.const\ (f\,(\mathbf{use}\ [1C]\ 0))\\ (f\,(\mathbf{use}\ [3C]\ 0)))(\mathbf{fn}\, x.x)$$

The second application nonetheless contributes to the type of f:

$$task\,[3C]\,int \to task\,[3C]\,int$$

The analysis result of this program is $3C$, whereas a programmer might expect $1C$. □

There are more problems with lambda-bound functions. The inability to give different types to different uses of a function can cause non-termination of the fixpoint iteration.

Example 2.5.10. In the following program, the argument f of *twice* is applied to its own result. In our current implementation, the worklist algorithm diverges when trying to analyse this program. The actual cost when running the program is $[4C]$.

$$\mathbf{let}\ twice = \mathbf{fn}\, f.\mathbf{fn}\, x.f\,(f\,x)\ \mathbf{in}\\ \mathbf{let}\ g = \mathbf{fn}\, t.t \gg t\ \mathbf{in}\\ twice\ g\ (\mathbf{use}\ [1C]\ 0)$$

Let us follow the type inference algorithm to understand the cause of the problem. f is a function, so it must have type $\alpha_1 \to \alpha_2$. Furthermore, f is applied to its own result, so **[t-app]** generates a subtype constraint $\alpha_2 <: \alpha_1$. The fact that *twice* is let-bound and therefore α_1 and α_2 are generalized does not help, because the problem emerges when the type and the constraint of *twice* get instantiated in the second let body. In the second

let body, g has type $task\ \beta\ int \to task\ (\beta \boxplus \beta)\ int$, with constraint $\beta \sqsupseteq [1C]$. Instantiating the type of *twice* for g, we get $\alpha_1 = task\ \beta\ int$ and $\alpha_2 = task\ (\beta \boxplus \beta)\ int$. Instantiating the constraint of *twice* for g, we get $task\ (\beta \boxplus \beta)\ int <: task\ \beta\ int$. According to the rules of subtyping this gives the constraint $\beta \sqsupseteq \beta \boxplus \beta$.

In the actual analysis algorithm, there are many more intermediate type- and annotation variables with trivial constraints like $\beta_7 \sqsupseteq \beta_8$ that connect them. We have omitted them here. What matters is that the resulting constraint set essentially looks like this:

$$\{\, \beta \sqsupseteq [1C],\ \beta \sqsupseteq \beta \boxplus \beta \,\}$$

This constraint set has a solution in $\mathbb{D}$, namely $[\beta \mapsto \infty]$. This solution cannot be computed by fixpoint iteration. The problem here is not the absence of widening. In fact applying widening, say after some fixed number of iterations, would lead to overly pessimistic results in other cases. These are cases where non-recursive constraints would reach a stable solution after many iterations. Example 2.5.8 demonstrates what can happen when widening is applied unnecessarily. The real problem here is the emergence of recursive constraints in the first place. Because f is lambda-bound, it gets the same type in both occurrences in $f\,(f\,x)$. The problem would not occur if we were able to give f a polymorphic type. This would require higher-ranked polymorphism, which is a topic for future work. With higher-ranked polymorphism, the resulting constraint set would look as follows.

$$\{\, \beta_1 \sqsupseteq [1C],\ \beta_2 \sqsupseteq \beta_1 \boxplus \beta_1,\ \beta_3 \sqsupseteq \beta_2 \boxplus \beta_2 \,\}$$

The second and third constraint come from the different instantiations of f's type. The least solution to this constraint set would have $\beta_3 \mapsto [4C]$, which is the result a programmer would expect for the program in this example. □

2.6 Future Work

There are three directions in which we would like to extend the system. They all revolve around making the analysis accessible to iTasks programmers.

First, we need a solution to the non-termination issue of example 2.5.10. A pragmatic approach, which is acceptable to programmers, as various discussions with our colleagues have suggested, is artificially limiting the height of the lattice $\mathbb{D}$. This can be done by parametrizing the analysis with a finite upper bound for each resource. Fixpoint iteration terminates when this upper bound is reached, which always happens in finitely many steps. In the real world any resource is anyway only available in a limited quantity. For a programmer it conveys the same information whether a task uses $n + 1$ or infinite units of a resource of which there are only n units. Limiting the height of the domain furthermore allows us to remove widening altogether, which also solves the problem that widening unnecessarily overestimates costs in situations like example 2.5.8.

Second, we would like to make the analysis applicable to real iTasks programs. As it stands, the analysis takes as input programs written in the language described in this chapter. The language has been designed to be a minimal variant of Clean and iTasks, but there are some features missing that are essential to everyday functional programming, most notably algebraic data types, pattern matching, and mutual recursion. Part of this second

point is integration with Tonic [Stutterheim et al., 2014], the system that can visualize iTasks programs and inspect them at run time. In particular, we would like Tonic to display predicted costs in static blueprints and actual costs in dynamic blueprints.

The third direction needed for real-world application is a concept for error reporting. The current implementation reports the overall cost of a program, but a programmer needs more information when that cost exceeds the available limit. This information is not as simple as pointing to an ill-typed expression in the program, because if there are many tasks whose costs together exceed the limit, there is not one obvious culprit. There is a technique called error slicing which identifies all parts of a program that contribute to an error message. Type error slicing [Haack and Wells, 2004] looks promising for our purpose from what we have seen so far, but further study is needed.

2.7 Related Work

For readers familiar with program analysis, and in particular type and effect systems, it should be obvious that we are following in the footsteps of Nielson and Nielson and their various co-authors. In particular, there is the textbook on program analysis [Nielson et al., 1999], and their triptych on polymorphic subtyping with Amtoft [Amtoft et al., 1997; Nielson et al., 1996b,a]. The paper on the IO monad in Haskell [Peyton-Jones, 2001] served as inspiration for various aspects of our dynamic semantics. Other papers whose approach to type and effect systems influenced our work are [Gedell et al., 2006], the work on exception analysis by Koot and Hage [2015], the usage analysis by Hage et al. [2007], and the security analysis by Weijers et al. [2014]. The distinction between consumable and reusable resources is common practice in the field of artificial intelligence and automated planning [Ghallab et al., 2004]. Papers that deal with the analysis of resource consumption of programs often focus on computational resources of the program itself like memory usage and execution time. Notable examples are [Vasconcelos and Hammond, 2003] and [Jost et al., 2010]. Kersten et al. [2014] have a resource analysis similar in spirit to ours. They focus on a single resource however, energy consumption of hardware components, and their language is imperative with first-order functions. The programming language Clean and the iTasks system, for which our analysis is ultimately designed, are described in the Clean language report [Plasmeijer and van Eekelen, 2002] and the paper by Plasmeijer et al. [2011].

Acknowledgements.

We would like to thank Ruud Koot, Jurriën Stutterheim, Tim Steenvoorden, Terry Stroup, Chris Elings, Fok Bolderheij, Marko van Eekelen, and Adelbert Bronkhorst for many hours of fruitful discussion.

3 The Sky is the Limit: Analysing Resource Consumption Over Time Using Skylines

In this chapter we extend the static analysis for costs of higher-order workflows with time. Costs are now maps from resource types to simple functions over time. We present a type and effect system together with an algorithm that yields safe approximations for the cost functions of programs.

3.1 Introduction

In chapter 2 we presented a static analysis for workflows that, given costs for basic tasks, yields a safe approximation of the cost of a whole program. Costs were maps from resource types to numbers, one number for each type of resource a program requires. In this chapter we refine the notion of cost by including time, in the sense that each basic task gets a duration, and its cost refers to this duration. The result of the extended analysis is not a single number, but a function over time. The main contribution of this chapter is the development of a two-dimensional cost model suitable for both the dynamic and static semantics (section 3.2). We also present a type system (section 3.3) and an implementation (section 3.4) for the analysis.

3.1.1 Basic Ideas

We would like to represent the cost of executing tasks by functions over time. Complex tasks are composed of simpler ones, and so should be their cost functions. By combining cost functions of simple tasks using operations that correspond to task composition, we obtain cost functions for complex tasks.

Example 3.1.1. Consider the two tasks of hosting a birthday party and a wedding, which both require chairs. Let's say the birthday party requires 20 chairs and takes five hours, while the wedding requires ten chairs and takes ten hours. We visualize these costs over time in diagrams called *skylines*. The skylines are shown in fig. 3.1. Hosting the birthday

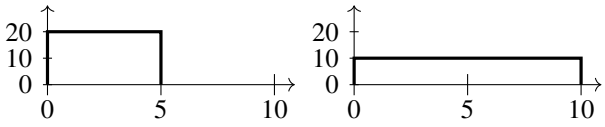


Figure 3.1: Cost skylines for a birthday and a wedding

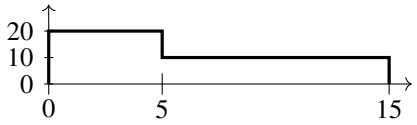


Figure 3.2: Cost skyline for a birthday *followed by* a wedding

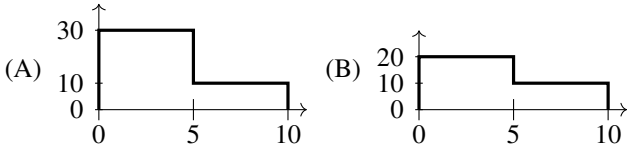


Figure 3.3: (A) Cost skyline for a birthday *and* a wedding (B) Cost skyline for a birthday *or* a wedding

$$
\begin{aligned}
e ::= {} & b \mid i \mid () \mid x \mid \mathbf{fn}\, x.e \mid \mathbf{fix} f x.e \mid e_1 e_2 \\
\mid {} & \mathbf{if}\, e_c\, \mathbf{then}\, e_t\, \mathbf{else}\, e_e \mid \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \mid e_1 \odot e_2 \\
\mid {} & \mathbf{use}\, [k, t]\, e \mid \mathbf{return}\, e \\
\mid {} & e_1 \,\&\, e_2 \mid e_1 \gg\!= e_2 \mid e_1 \gg e_2 \\
k ::= {} & nu \mid nu + k
\end{aligned}
$$

Figure 3.4: Syntax of the language

party first and immediately after it the wedding requires being able to supply 20 chairs for the first five hours and 10 chairs for the hours 5 to 15. The corresponding skyline is shown in fig. 3.2. Hosting the birthday party and the wedding at the same time requires 30 chairs for the first five hours, and 10 chairs for the remaining five hours. See fig. 3.3 (A). If, for whatever reason, either the wedding or the birthday takes place, then actually either 20 chairs for five hours or 10 chairs for ten hours are used. To be prepared for both situations, a host must calculate with 20 chairs for five hours and ten chairs for another five hours. See fig. 3.3 (B).

In this example, we considered chairs, which are a reusable resource. Combining skylines of consumable resources works differently, because those can only increase and never decrease. This is illustrated in example 3.2.17.

3.2 Syntax and Semantics

The only difference in the syntax of the language compared to chapter 2 is that the cost annotation of basic tasks now includes time. The syntax is defined in fig. 3.4, where the changed clause is highlighted. All basic tasks are represented by the **use** operator, where k denotes the cost of executing the task, and t its duration. Costs are given in a polynomial-like syntax where n is a natural number and u the unit of a resource. Durations t are given as

non-zero natural numbers. For example the expression "**use** $[2S + 3B, 20]$ 5" may denote a task that uses 2 screwdrivers, 3 bottles of wine, takes 20 minutes to execute, and yields the value 5. Costs, time, and resources are discussed in more detail in section 3.2.1.

3.2.1 A Domain for Representing Costs Over Time

In this section we develop a model for costs over time, suitable for both the dynamic and static semantics.

Definition 3.2.1. (Time) For our purposes, a *point* in time is an element of $\overline{\mathbb{N}}$. A *duration* is a non-zero natural number. □

Programmers can only give finite durations to basic tasks, but the analysis can give infinite durations to programs when needed.

Definition 3.2.2. (Skylines) The cost over time for a single resource is given by time series $\overline{\mathbb{N}} \to \overline{\mathbb{N}}$, called *skyline*. The set of all skylines is called $\mathbb{S}$. □

Definition 3.2.3. (Skyline ordering) Skylines are ordered pointwise. Let r, s be skylines.

$$r \sqsubseteq s \text{ iff } r(t) \sqsubseteq s(t) \text{ for all } t \in \overline{\mathbb{N}}$$ □

Definition 3.2.4. (Adding skylines) The sum of two skylines is defined pointwise.

$$(r + s)(t) = r(t) + s(t)$$ □

Definition 3.2.5. (Merging skylines) The merge of two skylines ($\sqcup$) is defined as their pointwise maximum.

$$(r \sqcup s)(t) = r(t) \sqcup s(t)$$ □

Appending skylines requires a bit more consideration. Tasks in the branches of a conditional can have different durations. For the overall skyline of a conditional, it is not sufficient to have the maximum length of the branches, as example 3.2.6 illustrates.

Example 3.2.6. The following program has a conditional with two cheap tasks of different length, followed by an expensive task.

```
(if  True
   then use (1R,2)  0
   else use (1R,4)  0
) ≫ use (3R,1)  0
```

A safe prediction must take into account that the spike can happen at time 2 or at time 4, see fig. 3.5 □

The set of possible start times of subsequent tasks grows with nested conditionals and conditionals in sequence. Our analysis keeps track of the earliest and latest times a task can end. All operations on costs must take this interval into account. End intervals belong to costs (definition 3.2.12), not to individual skylines. The end interval of a cost applies to all its skylines. When we talk about the end interval of a skyline, we mean the end interval of the cost of which the skyline is part of.

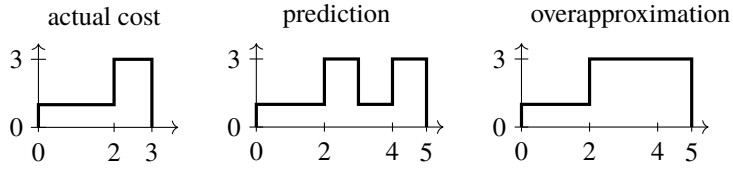


Figure 3.5: (left) The actual skyline of program 3.2.6. (middle) A prediction of the conditional must anticipate that either branch might be executed, followed by the remainder of the program. (right) It is safe to assume that the remainder can start anywhere between the earliest and latest end time of the branches.

Definition 3.2.7. An *end interval* is a pair $[x, y] : \mathbb{N} \times \overline{\mathbb{N}}$. □

Definition 3.2.8. (Operations on end intervals)

$$
\begin{aligned}
[x_1, y_1] \sqcup [x_2, y_2] &= [\min(x_1, x_2), \max(y_1, y_2)] \\
[x_1, y_1] + [x_2, y_2] &= [\max(x_1, x_2), \max(y_1, y_2)] \\
[x_1, y_1] +\!\!+ [x_2, y_2] &= [x_1 + x_2, y_1 + y_2]
\end{aligned}
$$

□

The operations $\sqcup$, $+$, $+\!\!+$ correspond to conditionals, parallel, and sequential composition of tasks respectively. The earliest possible end point of a *conditional* is the earliest one of the branches. The latest possible end point is the latest one. A *parallel* composition terminates when both branches terminate. The earliest possible end time of parallel tasks is therefore the maximum start point of their end intervals. The latest possible end time is the maximum end time of the tasks. The *sequential* composition of two tasks terminates after the second task terminates. The earliest possible end time is therefore the sum of the earliest end times of the operands, the latest possible end time is the sum of the latest end times of the operands.

Appending skylines needs to take end intervals into account. We make the simplifying but safe assumption that the second skyline can start anywhere in the end interval, see fig. 3.5 (right). Appending consumable and reusable skylines works slightly differently, as defined below.

Definition 3.2.9. (shift and bump) The function $\mathrm{shift}(s, n)$ shifts a skyline s to the right by n. The function $\mathrm{bump}(s, n, x)$ shifts a skyline s to the right by n and upwards by x.

Definition 3.2.10. (Appending reusable skylines) To append reusable skylines r and s, we shift s to every point where it could start and merge all possibilities. Formally, let e be the end interval of r.

$$r +\!\!+ s = r \sqcup \bigsqcup \{ \mathrm{shift}(s, n) \mid n \in e \}$$

□

Definition 3.2.11. (Appending consumable skylines) To append consumable skylines r and s, we bump s to every point where it could start and merge all possibilities. Let e be the end interval of r.

$$r +\!\!+ s = r \sqcup \bigsqcup \{ \mathrm{bump}(s, n, r(n)) \mid n \in e \}$$

□

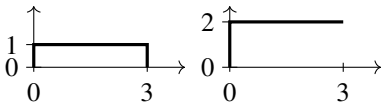


Figure 3.6: Basic skylines of $\mathrm{sky}(1R + 2C, 3)$

Definition 3.2.12. (Predicted cost) The *predicted cost* γ of a program is a tuple $\langle c, e \rangle$ where $c : U \to \mathbb{S}$ is a family of skylines, one for each resource in U, and e is an end interval. The end interval must be compatible with the skylines, which means it must end where the longest skyline ends. □

Definition 3.2.13. (Actual cost) The *actual cost* of a program uses the same construction as the predicted cost, but the end interval collapses to a single finite point. □

Definition 3.2.14. (Cost ordering) The ordering on costs must respect skylines and end intervals.

$$\langle c_1, e_1 \rangle \sqsupseteq \langle c_2, e_2 \rangle \iff e_1 \sqsupseteq e_2 \text{ and } s_1 \sqsupseteq s_2 \text{ for all skylines in } c_1, c_2.$$

The ordering on end intervals is interval inclusion. □

Definition 3.2.15. (Operations on costs) Operations on costs $\sqcup$, $+$, $+\!\!+$ are defined in terms of the respective operations on skylines and end intervals. □

3.2.2 Operational Semantics

The operational semantics with skylines is very similar to the one with scalar costs. We only discuss the differing parts.

Definition 3.2.16. (Basic skylines) The function *sky* takes a resource requirement and a duration d and returns a predicted cost with basic skylines. The end interval is the single point d. Basic skylines for each resource all have duration d and the height from their entry in the resource requirement. □

For example, the skylines of $\mathrm{sky}(1R + 2C, 3)$ look as in fig. 3.6. This figure also shows how we visualize consumable skylines. They never decrease, but always stay at the height they have reached.

The general-purpose semantics is identical to the one in fig. 2.2. The domain-specific semantics is given in fig. 3.7. The rules are mostly identical to the ones in fig. 2.3, except that operations on costs now refer to their counterparts for skylines. The only rules that differ are **[ds-use]** and **[ds-par-ret]**.

[ds-use] The rule specifies how a basic task is evaluated: The function *sky* creates basic skylines of duration d for all resources in k, and these skylines are then appended to the costs so-far γ.

[ds-par-ret] applies when both tasks in a process pool have been fully evaluated. The costs of each task have been tracked independently in the local costs γ_1 and γ_2. The idea is that these tasks run in parallel, so they cannot re-use each others resources. The local costs are added up and appended to the cost so far.

$$\textbf{[ds-use]} \quad \langle \textbf{use}\ [k,d]\ e,\ \gamma \rangle \twoheadrightarrow \langle \textbf{return}\ e,\ \gamma +\!\!+ sky(k,d) \rangle$$

$$\textbf{[ds-par-ret]}\ \langle pp(\textbf{return}\ () \,\&\, \textbf{return}\ (), \gamma_1, \gamma_2), \gamma \rangle \twoheadrightarrow \langle \textbf{return}\ (),\ \gamma +\!\!+ (\gamma_1 + \gamma_2) \rangle$$

Figure 3.7: Excerpt of the semantics for task expressions

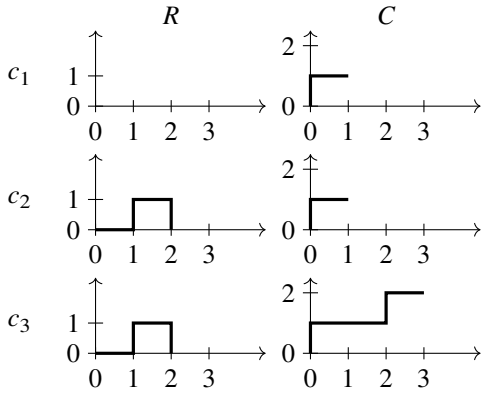


Figure 3.8: Progression of resource consumption of three tasks in sequence

Example 3.2.17. In this example we trace the reduction of a program, together with the progression of resource consumption. The program executes three tasks in sequence, where the middle task uses a different resource than the others. The end interval in the dynamic semantics always collapses to a single point, which is the elapsed time. For simplicity, we write configurations of the operational semantics as $\langle e, c, t \rangle$, where e is the expression to be reduced, c all skylines so-far, and t the single-point end interval.

$$\begin{aligned}
&\langle \textbf{use}\ [1C,1]\ 0 \gg \textbf{use}\ [1R,1]\ 0 \gg \textbf{use}\ [1C,1]\ 0, c_0, 0 \rangle \\
\twoheadrightarrow\ &\langle \textbf{return}\ 0 \gg \textbf{use}\ [1R,1]\ 0 \gg \textbf{use}\ [1C,1]\ 0, c_1, 1 \rangle \\
\twoheadrightarrow\ &\langle \textbf{use}\ [1R,1]\ 0 \gg \textbf{use}\ [1C,1]\ 0, c_1, 1 \rangle \\
\twoheadrightarrow\ &\langle \textbf{return}\ 0 \gg \textbf{use}\ [1C,1]\ 0, c_2, 2 \rangle \\
\twoheadrightarrow\ &\langle \textbf{use}\ [1C,1]\ 0, c_2, 2 \rangle \\
\twoheadrightarrow\ &\langle \textbf{return}\ 0, c_3, 3 \rangle
\end{aligned}$$

The progression of resource consumption is shown in fig. 3.8. First, the skyline for C takes a step (c_1). Then, the skyline for R takes a step (c_2), and finally the skyline for C takes another step (c_3). It is important to notice that the steps taken are not appended directly at the end of the skyline of the previous step, but at the point in time where the task is started. The semantics knows this because it tracks the elapsed time in the end interval. That is why the skyline for C stays at length 1 in c_2, and makes its next step at time 2 in c_3. □

Example 3.2.18. This example covers the reduction of a program with parallel tasks,

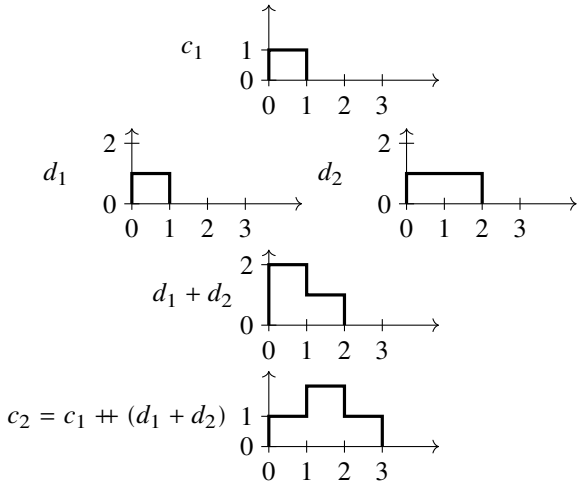


Figure 3.9: Progression of resource consumption of three tasks where two are in parallel

illustrating how process pools occur during execution. All three tasks in the program use the same reusable resource R.

$$\langle \textbf{use}\ [1R,1]\ 0 \gg (\textbf{use}\ [1R,1]\ () \ \&\ \textbf{use}\ [1R,2]\ ()), c_0, 0\rangle \tag{3.1}$$
$$\twoheadrightarrow \langle \textbf{return}\ 0 \gg (\textbf{use}\ [1R,1]\ () \ \&\ \textbf{use}\ [1R,2]\ ()), c_1, 1\rangle$$
$$\twoheadrightarrow \langle \textbf{use}\ [1R,1]\ () \ \&\ \textbf{use}\ [1R,2]\ (), c_1, 1\rangle$$
$$\twoheadrightarrow \langle pp(\textbf{use}\ [1R,1]\ () \ \&\ \textbf{use}\ [1R,2]\ (), \bot, \bot), c_1, 1\rangle \tag{3.2}$$
$$\twoheadrightarrow \langle pp(\textbf{return}\ () \ \&\ \textbf{use}\ [1R,2]\ (), d_1, \bot), c_1, 1\rangle$$
$$\twoheadrightarrow \langle pp(\textbf{return}\ () \ \&\ \textbf{return}\ (), d_1, d_2), c_1, 1\rangle \tag{3.3}$$
$$\twoheadrightarrow \langle \textbf{return}\ (), c_2, 3\rangle$$

The costs of this reduction are shown in fig. 3.9. In line (3.1), the first task is executed, creating the skyline c_1. In line (3.2), a process pool is created to prepare the execution of the parallel tasks. The process pool is initialized with empty initial costs for both tasks. In line (3.3), both tasks have been executed, with local skylines d_1 and d_2. In the last line, the process pool gets destroyed and the local costs get incorporated into the global cost with the formula $c_2 = c_1 \mathbin{+\!\!+} (d_1 + d_2)$.

3.3 Static Semantics

We now describe the type system for skylines. Before we discuss the typing rules, we have to look at the meaning of infinite costs and infinite time for skylines.

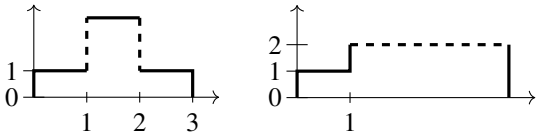


Figure 3.10: (left) A skyline of a reusable resource with infinite cost but finite length. (right) A skyline with infinite length but finite cost. Dashed lines are infinitely long.

3.3.1 Calculating With Infinity

At any point during *execution* of a workflow, the amount of expended resources so-far is finite, as is the elapsed time. The *analysis* however must be able to express that a workflow might have infinite cost or might take infinite time, or both. The following examples illustrate these kind of situations.

Example 3.3.1. The following program starts with a constant task, followed by a loop that runs n tasks in parallel, and finally ends with a constant task. The analysis cannot know how large the cost of the loop is going to be, so it must estimate it with infinity. The predicted skyline is shown in fig. 3.10 on the left side. The dashed part of the skyline stands for a cost of infinite height, but finite length.

```
let t = use [1R,1] 0 in
let loop = fix f n .
  if (n ≤ 1) then t else t & f (n−1)
in
t ≫ loop 5 t ≫ t
```

A loop whose length can only be predicted by infinity is one that runs a parametric number of tasks in sequence. The predicted skyline of such a program is shown in fig. 3.10 on the right side. The dashed part stands for a cost of finite height but infinite length. □

Example 3.3.2. Loops that require consumable resources often need predicted skylines with both infinite height and length. Consider the following program. It runs a task requiring a consumable resource a given number of times in sequence.

```
let repeat = fix f n. fn t.
  if n ≤ 0 then t else t ≫ (f (n−1) t)
in
repeat 3 (use [1C,1] 0)
```

The cost of this program cannot be statically bounded in height or in length, so the best approximation our analysis can make is a skyline that immediately jumps to infinity and stays there forever. □

3.3.2 The Annotated Type System

The architecture of constraint generation and solving is still the same as before, and not described here. The actual constraints and their semantics are different, defined as follows.

$$\textbf{[t-use]}\ \frac{C;\Gamma \vdash e : \hat{\tau} \qquad C \Vdash \beta \sqsupseteq \boxed{\mathrm{sky}(k,t)}}{C;\Gamma \vdash \boxed{\textbf{use}\ [k,t]\ e} : task\ \beta\ \hat{\tau}}\ \beta \text{ fresh}$$

$$\textbf{[t-bind]}\ \frac{\begin{array}{l} C;\Gamma \vdash e_1 : task\ \beta_1\ \hat{\tau}_1 \\ C;\Gamma \vdash e_2 : \hat{\tau}_1 \to task\ \beta_2\ \hat{\tau}_2 \\ C \Vdash \beta \sqsupseteq \boxed{\beta_1 +\!\!+ \beta_2} \\ \beta \text{ fresh} \end{array}}{C;\Gamma \vdash e_1 \gg\!= e_2 : task\ \beta\ \hat{\tau}_2} \qquad \textbf{[t-seq]}\ \frac{\begin{array}{l} C;\Gamma \vdash e_1 : task\ \beta_1\ \hat{\tau}_1 \\ C;\Gamma \vdash e_2 : task\ \beta_2\ \hat{\tau}_2 \\ C \Vdash \beta \sqsupseteq \boxed{\beta_1 +\!\!+ \beta_2} \\ \beta \text{ fresh} \end{array}}{C;\Gamma \vdash e_1 \gg e_2 : task\ \beta\ \hat{\tau}_2}$$

$$\textbf{[t-pp]}\ \frac{\begin{array}{l} C;\Gamma \vdash e_1 : task\ \beta_1\ \hat{\tau}_1 \\ C;\Gamma \vdash e_2 : task\ \beta_2\ \hat{\tau}_2 \\ C \Vdash \beta \sqsupseteq (\boxed{\beta_1 +\!\!+ \gamma_1}) + (\boxed{\beta_2 +\!\!+ \gamma_2}) \\ \beta \text{ fresh} \end{array}}{C;\Gamma \vdash pp(e_1 \,\&\, e_2, \gamma_1, \gamma_2) : task\ \beta\ \hat{\tau}_2}$$

Figure 3.11: The annotated type system

Definition 3.3.3. (Annotations) Annotations are expressions denoting costs. They are used in constraints to describe the cost behaviour of programs. Annotations are formed by the following grammar.

$$\varphi ::= k \mid \beta \mid \varphi_1 + \varphi_2 \mid \varphi_1 \nabla \varphi_2 \mid \boxed{\varphi_1 \sqcup \varphi_2} \mid \boxed{\varphi_1 +\!\!+ \varphi_2}$$

k stands for cost constants together with a duration, like $[2C+3R, 5]$. β stands for annotation variables. The operators $\sqcup, +, +\!\!+, \nabla$ stand for their respective operations on costs. □

Definition 3.3.4. (Constraints) The system still has the same two kinds of constraints, subsumption constraints ($<:$) and effect constraints ($\sqsupseteq$), formed by the following grammar. Annotations φ now refer to the ones of definition 3.3.3.

$$c ::= \hat{\tau}_1 <: \hat{\tau}_2 \mid \beta \sqsupseteq \varphi$$

3.3.2.1 Typing Rules

Typing judgements, as before, have the form $C;\Gamma \vdash e : \hat{\tau}$ where C is a constraint set, Γ a typing environment, e an expression and $\hat{\tau}$ an annotated type. The typing rules are almost the same as in fig. 2.5. Only the differing rules are shown and explained here. The differences are due to how costs are specified for basic tasks, and the different operation used for appending skylines.

[t-use] Basic tasks have the cost as specified in the program.

[t-bind] A sequential composition of two tasks has the cost of the right task appended to the cost of the left one.

$$\textbf{[t-seq]}\ \frac{\mathrm{T}_1 \qquad \mathrm{T}_2 \qquad C \Vdash \beta \sqsupseteq \beta_1 +\!\!+ \beta_2}{C; \emptyset \vdash \textbf{use}\ [1R, 1]\ () \gg \textbf{use}\ [2R, 1]\ () : \mathit{task}\ \beta\ ()}$$

where

$$C = \{\ \beta \sqsupseteq \beta_1 +\!\!+ \beta_2,\ \beta_1 \sqsupseteq \mathrm{sky}(1R, 1),\ \beta_2 \sqsupseteq \mathrm{sky}(2R, 1)\ \}$$

$$\mathrm{T}_1 = \textbf{[t-use]}\ \frac{C; \emptyset \vdash () : () \qquad C \Vdash \beta_1 \sqsupseteq \mathrm{sky}(1R, 1)}{C; \emptyset \vdash \textbf{use}\ [1R, 1]\ () : \mathit{task}\ \beta_1\ ()}$$

$$\mathrm{T}_2 = \textbf{[t-use]}\ \frac{C; \emptyset \vdash () : () \qquad C \Vdash \beta_2 \sqsupseteq \mathrm{sky}(2R, 1)}{C; \emptyset \vdash \textbf{use}\ [2R, 1]\ () : \mathit{task}\ \beta_2\ ()}$$

Figure 3.12: Typing derivation for a simple sequence

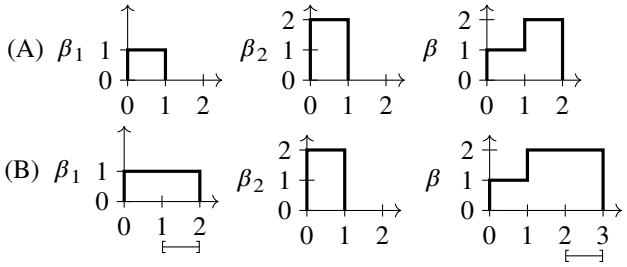


Figure 3.13: (A) Solution for the constraint set C of example 3.3.5 (B) Another solution, overapproximating β_1

[t-seq] ($\gg$) is a variant of ($\gg$=) that ignores the task value of its left-hand side. It has the same cost behaviour as ($\gg$=).

[t-pp] This rule states how to calculate the cost of two parallel tasks e_1 and e_2, whose execution to the current form already costed γ_1 and γ_2 respectively. β_1 and β_2 are the predicted costs of e_1 and e_2. The overall predicted cost of the process pool is the sum of the combinations of the actual costs so far and the predicted rest.

Example 3.3.5. This example demonstrates how the system predicts the cost of a simple sequential composition. Consider the following program.

$$\textbf{use}\ (1\mathrm{R},1)\ () \gg \textbf{use}\ (2\mathrm{R},1)\ ()$$

Figure 3.12 shows a typing derivation for this program, A solution of C is shown in fig. 3.13 (A), where β is an exact prediction of the actual cost of the program. Although this example only has a single skyline, we are implicitly talking about costs, which are families of skylines together with end intervals. End intervals are important, because fig. 3.13 (B) is also a solution to C, where β_1 is strictly above the required $\mathrm{sky}(1R, 1)$. To satisfy the ordering on costs it must respect the end interval of $\mathrm{sky}(1R, 1)$, which means it must include 1. The end intervals are drawn under the skylines. □

$$\mathcal{W}(\Gamma, \textbf{use}\ [k,d]\ e) = \langle task\ \beta\ \hat{\tau}_1, \theta_1, \{\ \beta \sqsupseteq sky(k,d)\ \} \cup C_1 \rangle \tag{3.4}$$
where β fresh
$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma, e)$

$$\mathcal{W}(\Gamma, e_1 \gg\!= e_2) = \langle task\ \beta\ ((\theta_4 \circ \theta_3)\alpha_2), \tag{3.5}$$
$\theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1$,
$(\theta_4 \circ \theta_3 \circ \theta_2)C_1 \cup (\theta_4 \circ \theta_3)C_2$
$\cup\ \{\ \beta \sqsupseteq ((\theta_4 \circ \theta_3)\beta_1) +\!\!+ (\theta_4\beta_2)\ \}\rangle$
where
$\langle \hat{\tau}_1, \theta_1, C_1 \rangle = \mathcal{W}(\Gamma, e_1)$
$\langle \hat{\tau}_2, \theta_2, C_2 \rangle = \mathcal{W}(\theta_1\Gamma, e_2)$
$\beta, \beta_1, \beta_2, \alpha_1, \alpha_2$ fresh
$\theta_3 = \mathcal{U}(\theta_2\hat{\tau}_1,\ task\ \beta_1\ \alpha_1)$
$\theta_4 = \mathcal{U}(\theta_3\hat{\tau}_2,\ (\theta_3\alpha_1) \to task\ \beta_2\ \alpha_2)$

Figure 3.14: Excerpt of the domain-specific part of algorithm $\mathcal{W}$

3.4 Implementation

The core of the analysis algorithm for skylines is mostly unchanged compared to the previous system. Most of the new code deals with skylines and operations on them. Unification and subsumption constraint solving are unchanged. Algorithm $\mathcal{W}$ still works the same, save for the places where it generates constraints about skylines. The biggest changes have been made to the effect constraint solver. In this section we only show the parts of $\mathcal{W}$ that differ from fig. 2.7, and the new constraint solver.

3.4.1 Algorithm W

Algorithm $\mathcal{W}$, as before, takes as input a program e and an environment Γ, and returns a triple of an annotated type $\hat{\tau}$, a substitution θ, and a set of constraints C. The returned results are such that if s is a solution of θC, and e is of type *task*, that is $\theta\hat{\tau} = task\ \beta\ \hat{\tau}_1$ for some β and $\hat{\tau}_1$, then $s(\beta)$ is an upper bound of the actual cost of e. The clauses of algorithm $\mathcal{W}$ different from fig. 2.7 are shown in fig. 3.14.

(3.4) This clause handles basic tasks. It generates a constraint that makes sure that the constraint solver takes the cost of the task into account.

(3.5) The clause for sequential task composition looks complex but holds no surprises. The left argument must be a task and the right argument a function that accepts the task's value. The function must yield a task. The clause generates a constraint that ensures that the cost of the overall expression is the append of the cost of the left and right arguments.

3.4.2 Subsumption Constraint Solving

The subsumption constraint solver is still the same as in our previous system, and not covered here. Subsumption constraint solving happens during and after algorithm $\mathscr{W}$, and its goal is to decompose subsumption constraints as far as possible to extract effect constraints according to the subtyping rules.

3.4.3 Effect Constraint Solving

The effect constraint solver for skylines differs from the one in section 2.4.3. The solver is a translation to functional code of the worklist algorithm found in chapter 6 in Nielson et al. [1999], with one modification. All skylines initially start at time 0 in their local coordinate system, and appending them to other skylines shifts them in time. Whether the information content of a single constraint has been fully incorporated into the final solution can no longer be determined by looking at the constraint in isolation. It is therefore not possible to determine whether a constraint has to be evaluated again in the worklist algorithm. To solve this problem we make use of the fact that in a complete lattice, these two formulations are equivalent: $x \sqsupseteq y \wedge x \sqsupseteq z \iff x \sqsupseteq y \sqcup z$.

Definition 3.4.1. (Combining constraints) Let C be a constraint set with possibly multiple constraints for each annotation variable β. The *combined constraints* C' have a single constraint for every β, such that

$$\begin{aligned} &\text{if} \quad \beta \sqsupseteq \varphi_1, \ \ldots, \ \beta \sqsupseteq \varphi_n \in C \\ &\text{then } \beta \sqsupseteq \varphi_1 \sqcup \ldots \sqcup \varphi_n \in C'. \end{aligned}$$

All constraints for every β are combined into a single constraint. □

The worklist algorithm for skylines takes as input the *combined constraints* of the constraints that algorithm $\mathscr{W}$ computes, and returns a solution of them. The algorithm is shown in fig. 3.15, where the difference with fig. 2.9 is highlighted.

(3.1) The current value for β just evaluates the constraint. This is possible because φ is the combination of all constraints regarding β.

(3.2) Determining whether φ has to be evaluated again now uses $\neq$, because as solving progresses, cost spikes tend to wander to the right to their final place. As they move further to the right, they disappear where they have been in the previous iteration. The new iteration is not strictly above the old iteration, yet the algorithm has learned something new.

Widening is the final piece of the puzzle, and the reason why the worklist algorithm always eventually terminates. Widening is a heuristic used by the effect constraint solver to compute solutions with infinity. The inputs to widening are two consecutive iterations of the same constraint. They correspond roughly to loop unrolling. The output of widening is a skyline that does not change in further iterations, representing an upper bound to the cost of the loop. Widening comes into play whenever the effect constraint solver encounters a constraint of the form $\varphi_1 \nabla \varphi_2$. Widening for skylines is more complicated than for scalar costs (definition 2.2.10). It needs to discard information to guarantee termination of the worklist iteration, while retaining enough information for the results to have some value. Widening applies to costs, and as such it must widen skylines and end intervals.

$$
\begin{aligned}
&\mathit{solve}(C) = \mathit{iterate}(\mathit{influences},\ C,\ \mathit{initSolution}) \\
&\quad \text{where} \\
&\quad \mathit{initSolution} = [\ \beta \mapsto \bot \mid \beta \in \mathit{FAV}(C)\ ] \\
&\quad \mathit{influences} = [\ \beta \mapsto \mathit{infl}'(\beta) \mid \beta \in \mathit{FAV}(C)\ ] \\
&\quad \mathit{infl}'(\beta) = \{\ \beta_1 \sqsupseteq \varphi \mid \beta_1 \sqsupseteq \varphi \in C, \beta \in \mathit{FAV}(\varphi)\ \} \\
&\mathit{iterate}(\_,\ \emptyset,\ s) = s \\
&\mathit{iterate}(\mathit{influences},\ \{\ \beta \sqsupseteq \varphi\ \} \cup \mathit{rest},\ s) = \\
&\quad \mathit{iterate}(\mathit{influences},\ \mathit{worklist}',\ s') \text{ where} \\
&\quad \mathit{newCost} = [\![\varphi]\!]s \qquad (3.1) \\
&\quad \mathit{dirty} = \mathit{newCost} \neq s(\beta) \qquad (3.2) \\
&\quad \mathit{worklist}' = \begin{cases} \mathit{rest} \cup \mathit{influences}(\beta) & \text{if } \mathit{dirty} \\ \mathit{rest} & \text{otherwise} \end{cases} \\
&\quad s' = s[\beta \mapsto \mathit{newCost}]
\end{aligned}
$$

Figure 3.15: The effect constraint solver

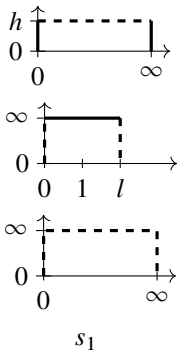
Definition 3.4.2. (Widening of end intervals)

$$
[x_1, y_1] \nabla [x_2, y_2] = \begin{cases} [x_1, \infty] & \text{if } y_2 > y_1 \\ [x_1, y_1] & \text{otherwise} \end{cases}
$$

The beginning of the end interval of a loop can never decrease, so the worst case is always the beginning of the left argument. If the end of the end interval grows, widening jumps to infinity. The other cases never occur and do not need to be considered. □

Definition 3.4.3. (Widening of reusable skylines) Let s_1, s_2 be two iterations of a reusable skyline s.

$$
s_1 \nabla s_2 = \begin{cases}
\text{if } s \text{ grows in length but not in height} \\
\text{if } s \text{ grows in height but not in length} \\
\text{if } s \text{ grows in length and height} \\
\text{otherwise}
\end{cases}
$$



Where h and l are the maximum height and length of s_1. We say s grows in length if s_2 is longer than s_1. Similarly s grows in height if the maximum height of s_2 is greater than the maximum height of s_1. □

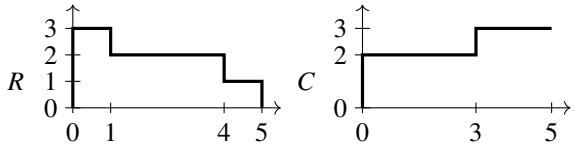


Figure 3.16: Costs for example 3.5.1

Definition 3.4.4. (Widening of consumable skylines) Let s_1, s_2 be two iterations of a consumable skyline s.

$$s_1 \nabla s_2 = \begin{cases} \text{impossible} & \text{if } s \text{ grows in length but not in height} \\ \infty,\ 0,\ 0\ 1\ l & \text{if } s \text{ grows in height but not in length} \\ \infty,\ 0,\ 0\ \infty & \text{if } s \text{ grows in length and height} \\ s_1 & \text{otherwise} \end{cases}$$

□

3.5 Discussion

In this section we discuss the analysis results of some example programs.

Example 3.5.1. (Exact predictions) When the control flow of a program does not depend on computations, the predicted cost matches the actual cost exactly, as in the following program.

let t1 = **use** [1R,1] 0 **in**
let t2 = **use** [1R+1C,4] 0 **in**
let t3 = **use** [1R+1C,3] 0 **in**
let t4 = **use** [1R+1C,2] 0 **in**
(t1 & t2) & (t3 ≫ t4)

This program is a mix of parallel and sequential compositions. The cost is shown in fig. 3.16.

Example 3.5.2. (Overapproximation by conditional) The following program demonstrates how the analysis overapproximates costs in both height and length.

let t1 = **use** [1R,1] 0 **in**
let t2 = **use** [3R,2] 0 **in**
let t3 = **use** [2R,1] 0 **in**
(**if** True **then** t1 **else** t2) ≫ t3

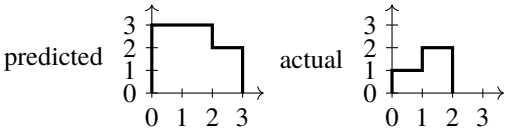


Figure 3.17: Costs for example 3.5.2

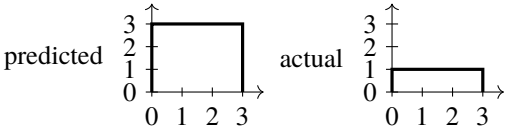


Figure 3.18: Costs for example 3.5.3

The overapproximation is caused by a conditional. There is a large cost in the else-branch which is not executed, but the analysis must take it into account. Task *t3* can start either at time 1 or at time 2, depending on which branch of the conditional is taken. Again, the analysis must anticipate both possibilities, and it does so by using end intervals. The actual cost is *t1* followed by *t3*, which is cheaper and does not take as long as the worst case. Predicted and actual costs are shown in fig. 3.17.

Example 3.5.3. (Overapproximation by poisoning) Poisoning happens when a variable is used in different contexts, which all influence its cost. There are several techniques in the literature to reduce poisoning, and we have included a couple of them, namely polymorphism, polyvariance, and subtyping. One source of poisoning that still exists are lambda-bound functions. Consider the following program.

```
let const = fn x . fn y . x in
(fn id . const (id (use [1R,3] 0)) (id (use [3R,3] 0))
) (fn x . x)
```

The identity function is lambda-bound and used in two different contexts. It is applied to an expensive task and a cheap task. The application to the expensive task is ignored by the function *const*, but it still influences the type of *id*. The type of *id* is not subject to polymorphism, because it is lambda-bound, and polymorphism only applies to let-bound variables. It is also not subject to subtyping, because subtyping only applies to the argument expression in function applications, not the function expression. The type of the identity function here is from expensive task to expensive task, so sending the cheap task through the identity function makes it expensive. Predicted and actual costs of this program are shown in fig. 3.18.

Example 3.5.4. (Widening in length) Another source of overapproximation are recursive functions. Consider the following program. The program uses *loop* to execute task *t1* three times, and then executes task *t2*.

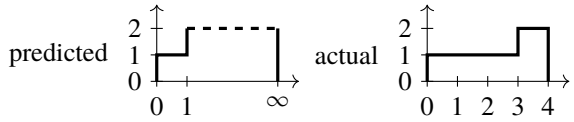


Figure 3.19: Costs for example 3.5.4

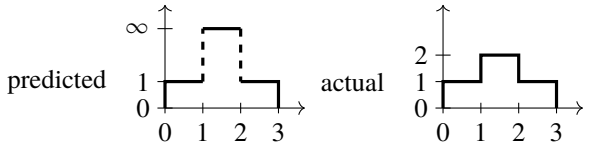


Figure 3.20: Costs for example 3.5.5

```
let t1 = use [1R,1] 0 in
let t2 = use [2R,1] 0 in
let loop = fix f x . if (x ≤ 1)
  then t1
  else t1 ≫ f (x−1) in
loop 3 ≫ t2
```

Our analysis cannot know how many times the loop will be executed, and has to approximate it with infinity. The analysis does see that *t1* is executed at least once. The predicted cost of the expression "loop 3" is therefore an infinite skyline of height 1, with end interval $[1, \infty]$. The end interval says that *t2* can start as early as time 1, but possibly also at any later time. The predicted and actual costs are shown in fig. 3.19.

Example 3.5.5. (Widening in height) The following program executes the task *t1* a number of times in parallel.

```
let t1 = use [1R,1] 0 in
let parallel = fix f x . if x ≤ 1
  then t1
  else  t1 & f (x−1) in
t1 ≫ parallel 2 ≫ t1
```

Our analysis cannot know how many times *t1* will be executed in parallel, so it has to overapproximate the cost with infinity. It does know that the duration of the parallel will always be 1. The predicted and actual costs are shown in fig. 3.20.

Example 3.5.6. (Limitations of the algorithm) There are programs that our implementation cannot handle, because they generate recursive constraints that are missed by widening, therefore causing the effect constraint solver to diverge. One example has already been shown in example 2.5.10. Another source of recursive constraints are polymorphic recursive functions that only later get instantiated to task types. Consider the following program.

```
let iterate = fix f n . fn g . fn x . if n ≤ 0
  then x
  else f (n−1) g (g x) in
let twice = fn t . t ≫ t in
iterate 2 twice (use [1R,1] 0)
```

The function *iterate* iterates its argument *g n*-times on *x*. When algorithm $\mathcal{W}$ typechecks *iterate*, which is polymorphic, it does not see any effect constraints, and therefore cannot apply widening. Widening as we have implemented it can only be applied to effect constraints. The subtyping constraints of *iterate* are recursive, and when they get instantiated to effect constraints in the call *iterate 2 twice*, we end up with un-widened recursive effect constraints. A solution to this problem would be to mark subtyping constraints as belonging to recursive functions, so that widening can be applied to them when they become effect constraints.

3.6 Conclusions

We have designed and implemented an annotated type system that can predict costs over time of a small iTasks-like workflow language. We do not have a correctness proof yet, and given the subtleties of operations on costs we imagine that it is a substantial amount of work. We do have an extensive test suite for the implementation, which includes many hand-crafted examples alongside automatically generated tests using Gast [Koopman et al., 2002]. The correctness of the implementation is tested by running the programs in an interpreter and checking that it never exceeds the prediction.

3.7 Future Work

There are a couple of ways to extend the system. One idea is to integrate the analysis into iTasks. At the moment, the analysis is defined on a condensed version of Clean and iTasks, but we would like to make the analysis available to iTasks programmers. This involves hooking into the compiler and supporting language features like data types, pattern matching, and mutual recursion. This line of work also includes integration with Tonic. Tonic [Stutterheim et al., 2014] is a system that can visualize iTasks programs graphically and allows inspecting their progress at run time. In particular we would like to display skylines alongside Tonic diagrams to show predicted costs next to control structure.

Another idea would be to trace points in a skyline back to the places in a program where they may come from. This would be particularly interesting in combination with Tonic. A user could interactively click on a point in a skyline, and the associated tasks in the Tonic diagram are highlighted.

Acknowledgements.

We would like to thank Jurriaan Hage, Tim Steenvoorden, Jurriën Stutterheim, Kelley van Evert, Terry Stroup, Bas Lijnse, and Ynske Aerts for many hours of fruitful discussion.

4 Skylines for Symbolic Energy Consumption Analysis

Energy consumption in embedded systems plays a large role as it has implications for the power supply and the batteries used. Programmers of these systems should consider how their programs control external devices, and where energy consumption hotspots lie. We present a static analysis to predict and visualize energy consumption of external devices controlled by programs written in a simple imperative programming language. Currently available energy consumption analysis techniques generate graphs over time, which makes it difficult to see from where in the source code the consumption originates. Our method generates graphs over source locations, called skyline diagrams*, showing the maximum power draw for each line of source code.*

Our method harnessess symbolic execution extended with support for controlling external devices. This gives accurate predictions and complete code path coverage, as far as the limits of computability allow. To make the diagrams easier to understand, we introduce a merge algorithm that condenses all skylines into a concise overview. We demonstrate the potential by analysing various example programs with our prototype implementation. We envision this approach being used to identify energy consumption hotspots of embedded systems during the design and development phase, in a less involved way than traditional approaches.

4.1 Introduction

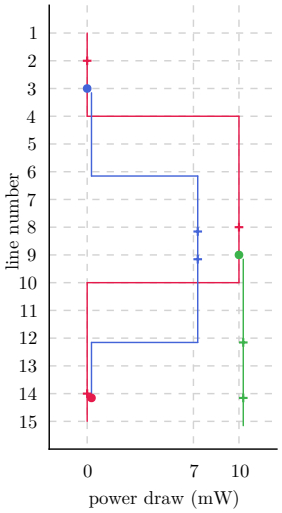
Software that controls hardware is found in many places, such as washing machines, smartphones, or self-driving cars. The software running in such devices is in charge of orchestrating the hardware components, like sensors, motors, displays, or radios. Formal analysis of such devices is hard, because hardware and software have to be analysed together. To optimize energy consumption of such devices, especially when they are battery-powered, it is useful for programmers to have a prediction of energy-behaviour of all the components when running their program. Simulation or actual measurement of running devices can give some insight, but only for one specific scenario and hardware configuration.

We develop a static analysis based on symbolic execution that can visualize the energy behaviour of all possible executions of a program at once. This allows programmers to quickly assess the energy impact of a change, already during development. Our method is parametrized with hardware models, so that programmers can swap components and explore different hardware configurations.

In the domain of embedded systems and control software, the energy use of the processor is sometimes negligible. We therefore limit our scope to the energy use of the hardware components controlled by the software. If desired, programmers can bypass this

```
1  int main() {
2    x = SENS.readTemp();
3    if( x ≤ 10 ) {
4      LED1.switchOn();
5    } else {
6      LED2.switchOn();
7    }
8    sleep(100);
9    if( x < 10 ) {
10     LED1.switchOff();
11   } else {
12     LED2.switchOff();
13   }
14   return 0;
15 }
```

(a)



(b)

Figure 4.1: (a) Sensor input controls a lamp (b) All possible runs of the program. One run does not end at power draw zero, which indicates a bug.

restriction by modelling the processor as a hardware component and switching between its power states explicitly with corresponding component calls. Modelling processors as hardware components is possible, because they often have approximately constant energy consumption. For example the popular ATmega328P, used on the Arduino UNO, has an amperage of 0.2 mA in Active Mode, 0.75 μA in Power-save Mode, and 0.1 μA in Power-down Mode [Mic, 2018].

We illustrate our approach with an example. The program in fig. 4.1a reads a sensor, and switches on either LED1 or LED2. It has a bug in line 9, where $<$ is used instead of $\leq$. There are three possible executions, one of which does not end with a power draw of zero, as there exists a sensor value where LED1 is not switched off. The skyline diagram in fig. 4.1b shows a merged view of the three executions. The horizontal axis shows power draw, and the vertical axis line numbers. Skylines that would be drawn on top of each other are shifted by a small offset. In this view, programmers can see that some component still consumes energy at the end of the function, and can start investigating the issue.

This chapter brings together two distinct lines of earlier work: the energy consumption analysis by van Gastel [2016] and the skyline diagrams by Klinik et al. [2017b]. Our contribution is threefold. First, we introduce a symbolic execution engine that tracks hardware state and works with the programming language SECA (Symbolic Energy Consumption Analysis). Second, we define visualization rules for the results of the symbolic execution as diagrams of power draw over points in the source code. Third, we define an algorithm to reduce the number of plotted graphs, hiding redundant information, to make the diagrams more concise. Our proof-of-concept implementation is available online [Klinik et al., b].

Our goal is to explore the idea of drawing energy skylines over source lines; not (yet) to make an industry-ready tool. To focus on this goal, our method considers a C-like language

that lacks the complexity of C itself. Likewise, the well-known problem of exponential state-space explosion, and reduction techniques that may be used to manage this problem, is not in our scope. Thus, we do not currently consider programs with thousands of lines.

4.2 Methodology

Given a program in the SECA language (section 4.3), our system performs symbolic execution to examine all possible execution paths. For each path, the symbolic execution engine tracks the power draw of all components that the program controls, resulting in a graph that relates program points to energy consumption (section 4.4). We call such graphs *skylines*. The result of symbolic execution is a set of skylines for every function, considering all calls to a function, across all execution paths. Our system then condenses these skylines into a summary of the energy behaviour of the program by *merging* common segments, to emphasize where skylines differ (section 4.5). The merged skylines are rendered as *skyline diagrams*, with line numbers on the vertical axis and power draw on the horizontal axis. The chapter ends with an analysis of a real-world example (section 4.6), a discussion of related work (section 4.7) and ideas for future work (section 4.8).

The domain we focus on is control software, whose main purpose is to control hardware like sensors or motors. It runs on embedded systems using low-powered microprocessors, which have a negligible energy use compared to the software-controlled hardware. Control software has two key characteristics. First, it has low algorithmic complexity. We aim to analyse programs that, for example, regulate a central heating installation, not those that calculate square roots. Second, it contains statements that interact with hardware components. These component calls are the focus of our system, as we seek to find how their invocation influences the energy behaviour of the program. If programs have parts with high algorithmic complexity, which would overextend the capabilities of symbolic execution, such parts could be hidden in library calls and left out of the analysis.

SECA represents the behaviour of hardware components in a model similar to the one by van Gastel [2016]; essentially a labelled transition system where every state has a power draw, and state changes can only be initiated by the code.

Resources other than power can also be modelled, as long as they can be summed up. The analysis sees resource consumption as a unitless number. We assume that components have rectangular power profiles, which means there is no ramp-up when switching them on.

Symbolic execution [King, 1976] is a program semantics that traces all possible program execution paths. Whenever a program asks for input, for example from a sensor or terminal, a symbolic input variable any_i is created. When conditionals are encountered, execution splits into two paths: one for evaluating the condition to true, one to false. Each path is coupled with the constraint on the symbolic inputs that must hold for this path to be followed. To illustrate the idea, consider the following program.

```
x = TEMP.readInt(); if (x <5) { y=7; } else { y=2*x+1; }
```

Symbolic execution results in two paths, one through the then- and one through the else-branch. The first one terminates with global state $[x \mapsto any_0, y \mapsto 7]$ and path constraint $any_0 < 5$. The second one terminates with $[x \mapsto any_0, y \mapsto 2any_0 + 1]$ and path constraint $\neg(any_0 < 5)$. Path constraints can be given to an SMT (Satisfiability Modulo Theories) solver, to prune infeasible paths, and calculate example values for the *any*s.

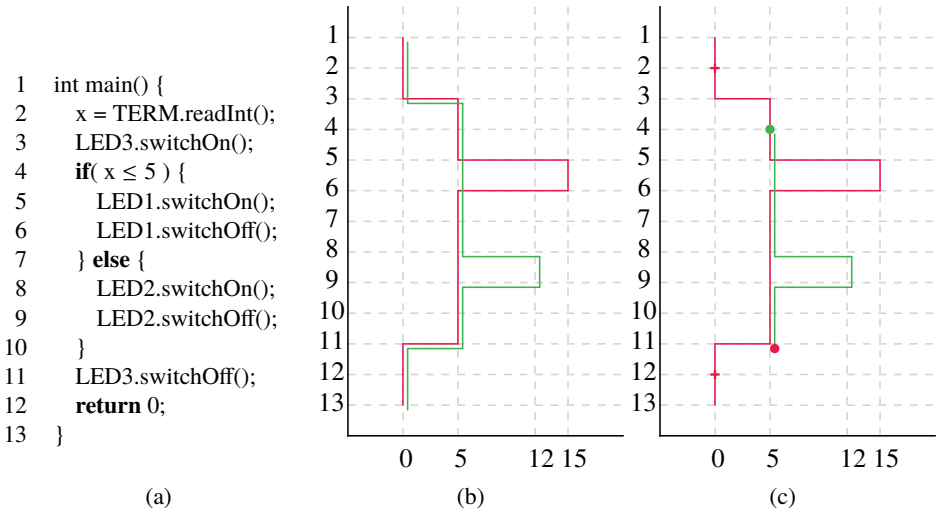


Figure 4.2: (a) A program with two execution paths (b) Its skylines (c) Its merged skylines

Symbolic execution does not terminate if there is a path that loops indefinitely. To bypass this problem, our system exits loops after a pre-defined number of iterations, and generates a warning. In such situations there could be paths whose energy usage is not reported, and hence the analysis is unsound. However, due to the nature of symbolic execution, all possible energy behaviours of a loop will often be discovered in less iterations than what is needed for the behaviours to occur in concrete execution. We expect situations with missed energy behaviours to be uncommon in typical programs.

In previous work [Klinik et al., 2017b] we analysed resource consumption over time. This has several advantages, but does not clearly show which parts of the program contribute to which parts of a skyline. Here, we give up the time aspect and instead relate resource use directly to lines in the source code. This requires certain coding conventions; for example, there may be only one non-trivial expression or statement per line, and closing braces must be on their own line. The results are diagrams with a natural control flow from top to bottom with occasional jumps, which clearly relate parts of the program to their energy consumption. Consider for example the program in fig. 4.2a. Symbolic execution results in the two skylines in fig. 4.2b, which show the hotspots in lines 5 and 8.

Symbolic execution results in many skylines, one for every execution path. These skylines often have identical parts, only differing after or up to a certain point. To emphasize where skylines differ, and de-emphasize where they are identical, we present an algorithm we call *merging*. Sometimes a skyline is equal to a second one for a few lines and then becomes equal to a third. This effect is common in loops, where a piece of code is executed repeatedly. The program in fig. 4.2a has two execution paths that only differ during the execution of the conditional (lines 4–10). Figure 4.2c shows its skylines after merging. Up to line 4 they are drawn as a single skyline. At line 4 is a *split point*, after which they are drawn separately. At line 11 they come together again, and continue so until the end of the function. Our system aims to give programmers an idea of the energy behaviour of

their programs, so that they gain insight where the hotspots lie. Merging comes at the cost of information loss about the exact number of runs, and losing the ability to fully trace individual runs. We argue that for our goal it is not required that skyline diagrams convey all information about all runs. Instead, we condense information with merging such that unexpected spikes can be clearly identified.

4.3 The SECA Language

SECA (Symbolic Energy Consumption Analysis) is a small imperative programming language. We designed SECA to look like a simple form of C, without features like raw memory access and pointer arithmetic. Such features complicate the analysis and are not the focus of this paper. We believe that with some engineering effort, the analysis can be extended to support the style of C programs common in embedded and safety-critical systems. SECA is a variant of ECA [van Gastel et al., 2015], which is itself a variant of Nielson's **While** [Nielson and Nielson, 1992]. SECA programs can control external hardware through *component calls*. For example, the component call LED1.switchOn() invokes the switchOn functionality of the component LED1. Component calls can perform I/O and have return values, but no arguments. While it would be simple to allow arguments to the component calls for the concrete component models, it would require significant changes to the symbolic component models. This would complicate the symbolic execution. For this paper we decided to keep this can of worms closed.

We make the following assumptions about SECA programs being well-formed. We provide no typing rules, but do require that programs are well-typed in the usual sense. We assume that all code paths of a function end in a return statement of the correct type, no references to undefined variables occur, and programs run on devices with all occurring hardware components. Void functions are allowed to end without return statements. In this case, the execution engine inserts an implicit return statement.

4.3.1 Syntax

The abstract syntax of SECA is shown in fig. 4.3. A program is a list of function- and global variable declarations. There must be one function main. Overlined symbols stand for lists of that symbol; for example $\overline{s}$ is a list of statements. Expressions are Boolean or integer constants, program variables, applications of binary or unary operators, function calls, and component calls. Operators are the usual Boolean connectives, comparisons and arithmetic. Function calls have a list of expressions, the parameters. Component calls have the form *name*.*function*() and invoke the specified function of the specified component. Statements are conditionals, while loops, assignments, returns, or expressions. While loops are annotated with a loop counter i, which the semantics uses to limit loop iterations, and to draw skylines differently in the first loop iteration. This is further discussed in section 4.3.2. The loop counter is not part of the concrete syntax, the programmer cannot access it, and it is initialised with zero. Assignments have a variable on the left-hand side and an expression on the right-hand side. Return statements end the current function call, and yield the given expression as the function's return value.

$$
\begin{aligned}
e &::= \mathsf{true} \mid \mathsf{false} \mid i \mid x \mid e\ op\ e \\
&\quad \mid un\ e \mid id(\overline{e}) \mid id.id() \\
op &::= \&\& \mid \mid\mid \mid \leq \mid < \mid + \mid - \mid \star \\
un &::= - \mid ! \\
s &::= \textbf{if}(e)\ \{\ \overline{s}\ \}\ \textbf{else}\ \{\ \overline{s}\ \} \\
&\quad \mid \textbf{while}(e)\ \{\ \overline{s}\ \}\ i \\
&\quad \mid x = e \mid \textbf{return}\ e \mid e \\
funcDef &::= id(\overline{x})\{\ \overline{s}\ \}
\end{aligned}
$$

Figure 4.3: Abstract syntax of SECA

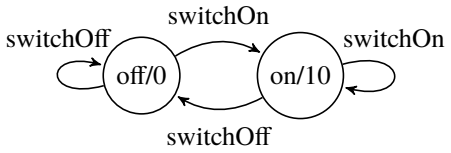


Figure 4.4: Hardware component model for an LED. In state *on* it has a power draw of 10, in state *off* a power draw of 0. The transitions correspond to component functions.

4.3.2 Semantics

SECA comes with four semantics, for different purposes. The *standard semantics* defines how programs are executed. The *energy-aware semantics* additionally traces the energy consumption during program execution in a skyline. The *symbolic execution semantics* executes all possible paths through a program. The *energy-aware symbolic execution semantics* traces all possible skylines a program can produce. The focus of this section is the last one; the others are formally defined in section 4.9. Below, we will informally discuss the energy-aware standard semantics, as it is a useful foundation to understand the energy-aware symbolic execution semantics.

To start, we must define the semantics of component calls. To analyse the energy consumption of programs, we need an estimation of how much energy their hardware components consume. Such an estimation is called a *hardware component model*, or *component model* for short. Component models are labelled transition systems, not necessarily finite, where each state has a power draw. Transitions are labelled with *component functions* (e.g. switchOn). Formally, a *hardware component model* $\langle S, L, \delta, o\rangle$ consists of a set of states S, a finite set of labels L, a transition function $\delta : L \times S \to IO(\mathbb{Z} \times S)$ and a power draw function $o : S \to \mathbb{N}$. A *configuration* of a model is an element of S: the current state. Every component has a start state. We borrow Haskell's notation $IO(\mathbb{Z} \times S)$ to indicate that to produce the return value $\mathbb{Z} \times S$, the function may perform arbitrary I/O. Input-producing hardware like sensors or terminals use the return value $\mathbb{Z}$ to return the input. Actuators like motors should return 0. The power draw function o specifies how much power the component consumes in each state. Take as an example the component model of an LED shown in fig. 4.4. LEDs have two states, *on* and *off*, and transitions *switchOn* and *switchOff* between them. In the *on* state an LED has a power draw of 10, in the *off* state it has a power draw of 0. The component functions do not return values.

SECA programs always run in contexts where a number of component models are present. Such contexts are called *component states*, or CStates for short. A CState is a partial mapping from names to configurations. If the CState contains an LED, say under the name of LED1, programs can contain the component calls LED1.switchOn() and LED1.switchOff().

The energy-aware semantics generates *skylines*. A skyline is a list of *segments*. A segment is either a start point $S(l, p)$ at line l and power draw p, a forwards line $F(l)$, a backwards jump $J(l)$, or an edge $E(p)$. Every skyline has exactly one start point, which must be its first segment. Other segments are interpreted relative to their predecessors. The y-axis of skylines refers to line numbers. Using line numbers to identify program

```
1  int main() {
2    x = 0;
3    while( x < 5 ) {
4      LED1.switchOn();
5      x = x + 1;
6    }
7    LED1.switchOff();
8    return 0;
9  }
```

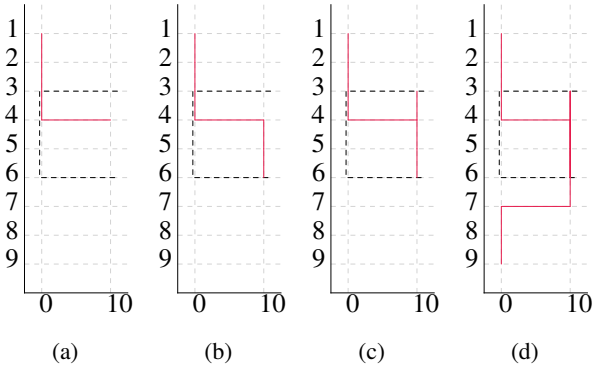


Figure 4.5: Stepwise construction of a skyline (a) After switching on LED1 (b) After the first loop iteration (c) After the second iteration (d) The final skyline

$$
\begin{array}{ll}
[S(1,0), F(2), F(3), F(4), E(10), & \text{(a)} \\
F(5), F(6), J(3), & \text{(b)} \\
F(4), E(10), F(5), F(6), J(3), & \text{(c)} \\
F(4), E(10), F(5), F(6), J(3), & \\
F(4), E(10), F(5), F(6), J(3), & \\
F(4), E(10), F(5), F(6), J(3), & \\
F(7), E(0), F(8), F(9)] & \text{(d)}
\end{array}
$$

Figure 4.6: Skyline fragments for fig. 4.5

points requires the source code to be formatted so that every skyline-producing program point is on its own line, to avoid segments being drawn over each other. This concerns the left-hand side of assignments, **if** keywords, **while** keywords, the closing brace of the body of while loops, **return** keywords, and the closing parentheses of function- and component calls. Expressions that contain function- or component calls as subexpressions should be split over multiple lines. Even with these restrictions, segments may end up on top of each other when the same lines of code are executed more than once.

We are now ready to look at the energy-aware standard semantics. Instead of formally defining it, we explain by example how the semantics executes a program and constructs its skyline on the way. The program in fig. 4.5 switches LED1 on five times in a loop, and then switches it off. The skyline fragments generated during execution are shown in fig. 4.6. Execution of this program starts in a CState where LED1 is in state *off*. Figure 4.5a shows the skyline just after executing line 4, where the LED has been switched on. Lines 2 and 3 do not change the power draw, which yields two forward segments from line 1 to 2 and from line 2 to 3. LED1 is switched on in line 4, which extends the skyline with a forward segment from 3 to 4, followed by a rising edge to power draw 10.

Figure 4.5b shows the skyline after one loop iteration, when the loop condition in line 3 has been executed a second time. Line 5 has caused a forward segment from 4 to 5. Execution of line 3 has caused a forward segment from the last statement of the loop in line 5 to the closing brace of the loop in line 6, followed by a backwards jump to line 3,

which is not visible in the diagram.

Figure 4.5c shows the skyline after two iterations. The second iteration starts at line 3 with power draw 10, and gives of three forward segments 3 to 4, 4 to 5, and 5 to 6. None of them change the power draw, as the LED is already on. These segments overlap with the segments of the previous iteration, and are drawn on top of each other. All subsequent iterations also generate identical segments.

After five loop iterations, the program exits the loop, with the skyline shown in fig. 4.5d. Switching off the LED generates a falling edge to power draw 0 in line 7. The return statement finally generates two forward segments, one for itself from 7 to 8 and one for exiting the function from 8 to 9.

4.4 Energy-Aware Symbolic Execution

The energy-aware symbolic execution semantics tracks path constraints and energy skylines for each execution path. The result is a set of skylines for each function together with their path constraints.

Under symbolic execution, expressions no longer evaluate to values, but to *symbolic values*. Symbolic values SVal are syntax trees whose leaves are constants or *symbolic inputs* any_i (variables that stand for an arbitrary integer). SVal is given by the grammar:

$$sv ::= \mathsf{true} \mid \mathsf{false} \mid i \mid any_i \mid sv\ op\ sv \mid un\ sv$$

In the symbolic semantics it is undesirable for component calls to perform I/O, because exploring all execution paths causes component calls to be executed multiple times. The symbolic semantics therefore uses component models where component calls return *symbolic values* with constraints. A *symbolic component model* $\langle S, L, \delta, o\rangle$ consists of a set of states S, a finite set of labels L, a transition function $\delta : L \times S \to \mathrm{SVal} \times \mathrm{SVal} \times S$, and a power draw function $o : S \to \mathbb{N}$. As opposed to the concrete models in section 4.3.2, δ cannot perform I/O. The first returned SVal of δ is typically a constant or symbolic input, and the second is a constraint on that input. For example, where the concrete model of TERM.readInt() asks for input and returns the user's answer, the symbolic model returns a fresh symbolic input any_j, with the constraint true, as this input can be any integer. A temperature sensor in a cold room can return a fresh any_j, together with a constraint $13 \leq any_j \wedge any_j \leq 17$. A symbolic LED returns constant 0, with the constraint true.

4.4.1 The Energy-Aware Symbolic Execution Semantics

We now study the algorithm that computes all possible executions of SECA programs, together with their corresponding skylines. The algorithm records the skylines of each function separately. This results in one skyline for each function call, for each execution path on which the call lies. The algorithm is defined by case distinction on the abstract syntax. We present the whole algorithm in figs. 4.7 and 4.8, a detailed description of the clauses follows. A formal definition of the semantics in the style common in programming language research, can be found in section 4.9. An implementation is available online [Klinik et al., b].

$$E : \mathbf{Expr} \times \Sigma \to \mathcal{P}(\mathbf{Val} \times \Sigma)$$

$$E[\![x]\!](\sigma) = \{\, \langle lookup(x,\sigma), \sigma\rangle \,\} \tag{4.1}$$

$$E[\![e_1\ op\ e_2]\!](\sigma) = \{\, \langle v_1\ op\ v_2, \sigma''\rangle \mid \langle v_1, \sigma'\rangle \in E[\![e_1]\!](\sigma), \langle v_2, \sigma''\rangle \in E[\![e_2]\!](\sigma') \,\} \tag{4.2}$$

$$E[\![un\ e]\!](\sigma) = \{\, \langle un\ v, \sigma'\rangle \mid \langle v, \sigma'\rangle \in E[\![e]\!](\sigma) \,\} \tag{4.3}$$

$$E[\![f(\overline{e})]\!](\sigma) = \{\, \langle lookup(\#return, \sigma'''), \sigma'''[pc \mapsto \sigma.pc]\rangle \mid \langle \overline{v}, \sigma'\rangle \in \overline{E}[\![\overline{e}]\!](\sigma), \sigma'' = call[\![f(\overline{v})]\!](\sigma'[pc \mapsto []]), \sigma''' \in X(\sigma'') \,\} \tag{4.4}$$

$$E[\![c.f()]\!](\sigma) = \{\, \langle v, \sigma'\rangle \,\} \tag{4.5}$$

where

$\sigma' = \sigma[cstate \mapsto cstate', sky \mapsto sky', \varphi \mapsto \varphi']$

$\langle v, \psi, s_c'\rangle = \delta_c(f, s_c)$

$cstate' = \sigma.cstate[c \mapsto s_c']$

$s_c = \sigma.cstate[c]$

$p = powerDraw(cstate')$

$l = lineOfCompCall$

$sky' = \sigma.sky +\!\!+ [F(l), E(p)]$

$\varphi' = \sigma.\varphi \wedge \psi$

$$call : \mathbf{Expr} \times \Sigma \to \Sigma$$

$$call[\![f(\overline{v})]\!](\sigma) = \sigma' \tag{4.6}$$

where

$\sigma' = \sigma[env \mapsto env', sky \mapsto sky', pc \mapsto \overline{s}, stack \mapsto stack']$

$env' = [\overline{x} \mapsto \overline{v}]$

$p = powerDraw(\sigma.cstate)$

$l = lineOfOpeningBrace$

$sky' = [S(l,p)]$

$\overline{s} = functionBody[\![f]\!]$

$stack' = push(\sigma, stack)$

Figure 4.7: The function E for expressions

$$X : \Sigma \to \mathcal{P}(\Sigma)$$

$$X(\sigma) = \begin{cases} \{\, \sigma \,\} & \text{if } \sigma.pc = [] \\ \bigcup \{\, X(\sigma') \mid \sigma' \in S[\![s]\!](\sigma[pc \mapsto rest]) \,\} & \text{if } \sigma.pc = [s] +\!\!+ rest \end{cases} \tag{4.2}$$

$$S : \mathbf{Stmt} \times \Sigma \to \mathcal{P}(\Sigma)$$

$$S[\![x = e]\!](\sigma) = \{\, assign(x, v, \sigma') \mid \langle v, \sigma'\rangle \in E[\![e]\!](\sigma[sky \mapsto \sigma.sky +\!\!+ [F(l)]]) \text{ where } l = lineNumberOfAssignment \,\} \tag{4.7}$$

$$S[\![\mathbf{if}(e)\ \{\,\overline{s}_1\,\}\ \mathbf{else}\ \{\,\overline{s}_2\,\}]\!](\sigma) = \bigcup \{\, \{\, \sigma'[pc \mapsto pc_1, \varphi \mapsto \varphi_1], \sigma'[pc \mapsto pc_2, \varphi \mapsto \varphi_2] \,\} \mid \langle v, \sigma'\rangle \in E[\![e]\!](\sigma[sky \mapsto \sigma.sky +\!\!+ [F(l)]]) \tag{4.8}$$

where

$l = lineOfIfKeyword$

$\varphi_1 = \sigma'.\varphi \wedge v$

$\varphi_2 = \sigma'.\varphi \wedge \neg v$

$pc_1 = \overline{s}_1 +\!\!+ \sigma'.pc$

$pc_2 = \overline{s}_2 +\!\!+ \sigma'.pc$

$$S[\![\mathbf{while}(e)\ \{\,\overline{s}\,\}\ i]\!](\sigma) = \bigcup \{\, \{\, \sigma'[pc \mapsto loop, \varphi \mapsto \varphi_1], \sigma'[\varphi \mapsto \varphi_2] \,\} \mid \langle v, \sigma'\rangle \in E[\![e]\!](\sigma[sky \mapsto sky']) \tag{4.9}$$

where

$\varphi_1 = \sigma'.\varphi \wedge v$

$\varphi_2 = \sigma'.\varphi \wedge \neg v$

$$sky' = \begin{cases} \sigma.sky +\!\!+ [F(l)] & \text{if } i = 0 \\ \sigma.sky +\!\!+ [F(m), J(l)] & \text{otherwise} \end{cases}$$

$loop = \overline{s} +\!\!+ [\mathbf{while}(e)\{\overline{s}\}(i+1)] +\!\!+ \sigma'.pc$

$l = lineOfWhileKeyword$

$m = lineOfClosingBrace$

Figure 4.8: The function X to execute whole programs and S for statements (continued on the next page)

$$
\begin{aligned}
&S[\![\textbf{return } e]\!](\sigma) = \\
&\quad \{\, registerSkyline(f, sky'', \sigma'') \\
&\quad \mid \langle v, \sigma' \rangle \in E[\![e]\!](\sigma[sky \mapsto sky']) \\
&\quad \text{where} \\
&\quad sky' = \sigma.sky +\!\!+ [F(l)] \\
&\quad sky'' = \sigma'.sky +\!\!+ [F(m)] \\
&\quad sky'_c = \sigma_c.sky +\!\!+ [F(n), E(p)] \\
&\quad \langle \sigma_c, stack_c \rangle = pop(\sigma.stack) \\
&\quad \sigma'' = \sigma'[env \mapsto \sigma_c.env[\#return \mapsto v] \\
&\qquad , pc \mapsto \sigma_c.pc, sky \mapsto sky'_c, stack \mapsto stack_c] \\
&\quad l = lineOfReturnKeyword \\
&\quad m = lineOfClosingBrace \\
&\quad n = lineOfCallSite \\
&\quad p = powerDraw(\sigma') \,\}
\end{aligned}
\tag{4.10}
$$

$$
S[\![e]\!](\sigma) = \{\, \sigma' \mid \langle v, \sigma' \rangle \in E[\![e]\!](\sigma) \,\}
\tag{4.11}
$$

Figure 4.8 (cont.): The function S for statements

Figures 4.7 and 4.8 show pseudocode for the functions E and S that compute symbolic skylines for expressions and statements respectively. We elaborate on some of the clauses below. Application of a function to syntactic arguments is denoted with double brackets $[\![-]\!]$, which have no further special meaning. A program state $\sigma \in \Sigma$ is a record with all information needed to execute a statement. It contains the values of local program variables *env* and global program variables *genv*, the current skyline *sky*, the current path constraint φ, the program counter *pc* (a list of statements to be executed after the current statement), the function call stack *stack*, and the CState *cstate*. The helper functions *lookup* and *assign* (not shown here) ensure that the scoping rules are respected, which means they prefer variables in *env* over *genv*.

Each clause of the semantics specifies how a single statement or expression together with a given program state produces the set of all possible immediate successor program states. Hence, a statement can be seen as a state transformer $\Sigma \to \mathcal{P}(\Sigma)$. To compose two functions of this type, we need glue code that applies the second function to every result of the first function. This is implemented by the function X in fig. 4.7, which executes whole programs.

4.4.2 Evaluation of Expressions

The evaluation function E (fig. 4.7) takes an expression e and a program state and returns the set of all possible values that e can evaluate to, together with the updated program states.

Clause (4.1) A variable x is evaluated by looking up its value in the environment. Only a single result is produced.

Clause (4.2) To evaluate a binary operator, all possible values v_1 for e_1, and all possible values for e_2 are calculated. The evaluations of e_2 happen in the result states of the evaluations of e_1. The result is the set of the symbolic values $v_1 op v_2$ for all combinations (v_1, v_2). These values are subject to constant folding, for example 1 + 2 becomes 3. This is not shown here.

Clause (4.3) The result of evaluating a unary operator is the operator applied to all values that the argument can evaluate to.

Clause (4.4) To evaluate a function call, first all arguments are evaluated. This is done by the sequential extension $\overline{E}$, which chains the state through the evaluation of the argument vector $\overline{e}$ and results in the set of all possible value vectors $\overline{v}$. For each vector $\overline{v}$, the helper function *call*, described below, prepares the function call, and X executes it. This execution will eventually end with a return statement. The return statement restores the program state so that σ''' can be used as the result state at the call site. The resulting value of the function call is the value of the #return register in σ'''.

Clause (4.5) To evaluate a component call, first the transition function δ_c of the component c is invoked, with the function name f and the component's current state s_c as arguments. This yields a return value v, a constraint ψ on v, and a new component state s'_c. The total power draw p after the call is computed. The skyline is extended with a forward segment $F(l)$ to the location l of the call site, followed by an edge $E(p)$ to the new power draw. The result of evaluating $c.f()$ is the return value v of δ_c, together with the updated program state.

Clause (4.6) The helper function *call* prepares the program state for execution of the function. It first initializes the environment env' for the function body with the actual arguments. It then starts a new skyline for the call to f with the current power draw p at the location l of the opening brace of the function definition of f. It uses the function body $\overline{s}$ as program counter, and creates a new stack frame for the function call.

4.4.3 Execution of Statements

The function X (fig. 4.8) recursively executes all statements of the program counter $\sigma.pc$, and collects the results. Execution of a SECA program starts in a program state that contains the body of the main function as program counter.

The function S (fig. 4.8) executes a single statement in a given program state, and returns all possible successor program states.

Clause (4.7) Assignments are executed by first extending the current skyline to the line of the assignment. Then e is evaluated to all its possible values, and the final results are all successor states where x has value v. Expressions can have side effects, so the successor states may have different skylines.

Clause (4.8) Execution of conditionals starts with extending the current skyline with a forward segment $F(l)$ to the line l of the *if* keyword. Then, all possible values v of the condition e are computed. This results in paths into both branches, for each v. The path constraint for the *then* branch is extended with v, for the *else* branch with $\neg v$. The program counter pc_1 specifies that first the statements $\overline{s}_1$ of the then-branch are

executed, and after that the original continuation $\sigma'.pc$. Similarly for the else-branch. If the SMT solver sees that φ_1 or φ_2 is unsatisfiable, their states are pruned (not shown).

Clause (4.9) For the first iteration of while loops, we need to generate a different skyline than for subsequent iterations. The first loop iteration can be recognized by the loop counter i being 0. If this is the case, the current skyline comes from outside the loop body, and we extend it with a forward segment $F(l)$ to the line of the *while* keyword. Otherwise, the current skyline comes from inside the loop body, and is extended with a forward segment $F(m)$ to the line m of the closing brace, followed by a backwards jump $J(l)$ to the beginning of the loop. In both cases, the condition e is evaluated to all possible values v. For every v, we generate two continuations: one for entering the loop with path constraint $\sigma'.\varphi \wedge v$ and one for exiting the loop with constraint $\sigma'.\varphi \wedge \neg v$. The program counter *loop* for entering the loop consists of the loop body $\overline{s}$, followed by the loop itself with incremented loop counter, then by what comes after the loop $\sigma'.pc$. The program counter for exiting needs no change, as $\sigma'.pc$ already contains the instructions following the loop. Our implementation uses the loop counter to bound the number of iterations. This is not shown here.

Clause (4.10) The clause for return statements has to deal with two different skylines: the one from the function that is about to return, and the one from the caller. Let f be the name of the current function. To execute a return statement, the current skyline is first extended with $F(l)$ to the location l of the return keyword. Then, the returned expression is evaluated. Next the skyline is extended with $F(m)$, to the location m of the closing brace of the function body. Then, the program state from before the function call is restored, but updated with all the changes made by f. For this, the topmost element σ_c of the call stack is removed; this is the program state of the caller. A new state σ'' is constructed, which the caller should use to resume execution; σ'' has the caller's original *env*, but with the #return register holding the return value. The program counter and call stack are restored to the ones from before the call. The caller's skyline $\sigma_c.sky$ is extended with a forward line $F(n)$ to the call site, and an edge $E(p)$ to the power draw p. Finally, the skyline of the function call is recorded in the list of all skylines of f. This is done with the function *registerSkyline*, which stores the given skyline in the given state, and returns the thus updated state.

Clause (4.11) Expressions can occur as statements. The expression's value is discarded, and only the state after evaluation is kept.

4.5 Merging Skylines

The skylines of a program often have many identical parts. Take for example the program in fig. 4.9. Most of its execution paths have identical energy behaviour. To merge skylines, we use a three-phase algorithm: *preparation*, *merging* and *finalization*. It is executed independently for every function.

In the preparation phase, all skylines are split into *fragments* and stored in an array, giving each a unique index. Fragments represent single horizontal or vertical line segments with explicit start and end points. Every fragment has a set of continuations: indexes of the fragments that follow it. Merging deletes explicit jumps $J(l)$: they are kept implicitly as fragments whose start point does not coincide with the end point of their predecessor. Ini-

```
1   int main() {
2     x = TERM.readInt();
3     while(0 ≤ x && x ≤ 2){
4       y = TERM.readInt();
5       while(0 ≤ y && y ≤ 2){
6         LED1.switchOn();
7         LED1.switchOff();
8         y = y + 1;
9       }
10      LED2.switchOn();
11      LED2.switchOff();
12      x = x + 1;
13    }
14    return 0;
15  }
```

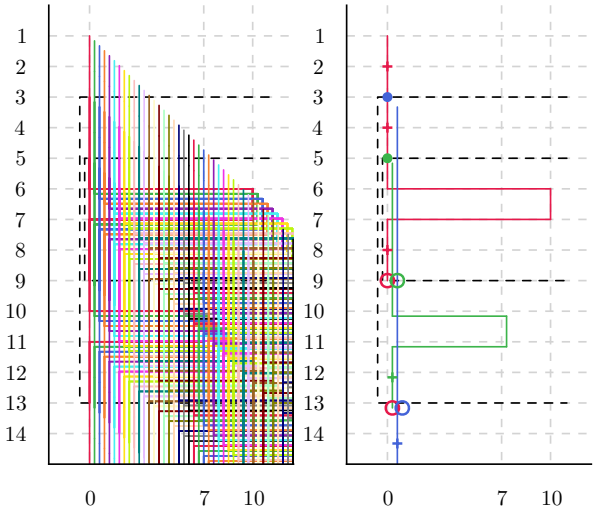


Figure 4.9: A program with many execution paths, and its unmerged and merged skylines

tially each fragment has at most one continuation, but more may be added later. Preparation is shown in fig. 4.10.

The merging phase takes the fragments created by preparation and deletes identical fragments. Whenever two fragments indexed i and j are equal, we can merge them by first combining their continuations, and then replacing all occurrences of j in continuations of other fragments by i. This is formally described by fig. 4.11.

Finally, in the visualization phase, fragments are grouped into skylines, by assigning the same colour to directly connected fragments. Figure 4.12 implements this. It then assigns a small diagonal offset to each colour group (not shown here), to avoid drawing lines on top of each other. Between two consecutive horizontal lines, a + indicates that a statement was executed at that point. Continuations are drawn as coloured bullets or circles: if $j \in frags[i].conts$ and $colour[i] \neq colour[j]$, then if $frags[i].end = frags[j].start$ then a bullet in $colour[j]$ is drawn at the end of fragment i. Otherwise, the continuation is a jump backwards; this is indicated by drawing an open circle in $colour[j]$ at the end of fragment i. Dotted lines in the diagram indicate the beginning and end points of loops.

4.6 A Real-World Example: Line-Following Robot

In this section we demonstrate how to apply our analysis to an existing real-world example, written in C. The program is simple enough that there is no potential for energy consumption optimization, but nonetheless our analysis gives insight into the program's energy behaviour. We chose a random "simple line follower" project from the Arduino project database [Arduino AG]. This robot has two motors and two sensors, and uses them to follow a black line on the floor. It works as follows. The sensors are positioned to the left and right of the line. If only the left sensor sees the line, the robot turns left. Symmetrically for the

```
procedure PREPARE(skies(f))
  // input: all skylines of function f
  // output: frags, an array of fragments,
  // each with at most one continuation
  Nfrags ← 0
  for all Skyline sky ∈ skies(f) do
    // all skylines begin with S(l,p)
    let ⟨l, p⟩ be such that sky[1] = S(l, p)
    for i ← 2 to length(sky) do
      sky[i] is either F(l′) or J(l′) or E(p′)
      in the first two cases, let p′ = p
      in the last case, let l′ = l
      if sky[i] is F(l′) or E(p′) then
        Nfrags ← Nfrags + 1
        frags[Nfrags].start ← ⟨l, p⟩
        frags[Nfrags].end ← ⟨l′, p′⟩
        frags[Nfrags − 1].conts ← { Nfrags }
      end if
      ⟨l, p⟩ ← ⟨l′, p′⟩
    end for
    // last fragment has no continuation
    frags[Nfrags].conts ← ∅
  end for
end procedure
```

Figure 4.10: Initializing the *frags* array

```
procedure MERGE(frags, Nfrags)
  // frags, Nfrags as produced by prepare
  // output: modified frags with equal
  // fragments merged
  for i ← 1 to Nfrags − 1 do
    if frags[i] = null then continue
    for j ← i + 1 to Nfrags do
      if frags[j] = null
      or frags[i].start ≠ frags[j].start
      or frags[i].end ≠ frags[j].end
        then continue
      frags[i].conts ←
        frags[i].conts ∪ frags[j].conts
      frags[j] ← null
      for k ← 1 to Nfrags do
        if frags[k] ≠ null ∧
          j ∈ frags[k].conts then
          frags[k].conts ←
            (frags[k].conts \ { j }) ∪ { i }
        end if
      end for
    end for
  end for
end procedure
```

Figure 4.11: Merging fragments

right sensor. If neither sensor sees the line, the robot moves forward. If both sensors see the line, the robot stops. The code has potential for refactoring, as it contains unnecessary repetition. However our goal was not to find the most elegant line follower robot, but to apply our method to a real-world example.

The original source code, written in C, is almost valid SECA. We made two changes to the code for our parser to accept it. First, we defined the constants `LOW` and `HIGH`, and the function `delay`, which for our purpose is empty. Second, we replaced the statements that write to output pins and read from input pins with component calls. Figure 4.13 shows an excerpt of the code after these adjustments. We then created the component models for motors and sensors in the source code of our analysis engine. The simulated motors have three states, *forward*, *backward*, and *stop*, and corresponding component calls. In the forward and backward states, motors have a power draw of 750mW. The sensors have no power draw, and their `read` component call returns a symbolic value in $\{0, 1\}$.

The analysis result for the robot is shown in fig. 4.14 as skyline diagrams for the functions *loop* and *MoveForward*. The diagram for *MoveForward* shows that the function has two behaviours, one where the power draw increases in two steps from 0 to 1500 mW, and one where the power draw stays constant at 1500 mW. The functions *TurnLeft* and *TurnRight*, not shown here, look similar. The function *Stop*, not shown, has the opposite

```
procedure COLOURIZE(frags)
  for i ← 1 to Nfrags do
    if (i > 1) ∧ (i ∈ frags[i − 1].conts) ∧
    (frags[i − 1].end = frags[i].start) then
      colour[i] ← colour[i − 1]
    else
      colour[i] ← a fresh colour
    end if
  end for
end procedure
```

Figure 4.12: Assigning colours

behaviour: the power draw decreases in two steps. The function *loop* has many behaviours, depending on the executed conditional. There are the cases where the power draw stays 0 or 1500 mW, or it can increase in lines 10, 18, or 22, or it can decrease in line 14.

Symbolic execution has high computational complexity, and our algorithm took longer to analyse the original robot program than we were willing to wait. The source of the high complexity lies in the function *loop*. We had to set the iteration limit to 2 for the analysis to terminate within 20 seconds on a ten year old laptop. The merged diagram would not change with more iterations. The high complexity occurs because firstly the four conditionals can be entered independently, and secondly the sensors are read in each condition, making the conditionals not mutually exclusive. This results in 16 possible executions of the function. Refactoring the program either by reading the sensors once at the start of loop() or by nesting the conditionals, reduces the number of possible executions to 4, making the analysis terminate in 2.4 seconds with iteration limit 2. An improved version of the robot program together with its skyline diagrams can be found on the project website [Klinik et al., a].

4.7 Related Work

Directly related are the second author's previous works [van Gastel et al., 2015; Kersten et al., 2014] describing static energy analyses for the language ECA. The first derives energy bounds for a specific input scenario; the second is a symbolic analysis that over-approximates all possible paths. These works do not use skylines. They do use hardware models that also support incidental one-time energy costs. This incidental energy cost can be useful for approximating energy consumption that varies over time. Also closely related is the first author's previous work [Klinik et al., 2017b], which introduces skylines, but also overapproximates since it estimates resource use over time.

Most publications on energy efficiency of software approach the problem on a high level, defining programming and design patterns for writing energy-efficient code, for example [Albers, 2010; Ranganathan, 2010; Saxe, 2010]. Cohen et al. [2012] and Sampson et al. [2011], divide a program into *phases* describing similar behaviour. Based on the behaviour of the software, design-level optimizations are proposed to achieve lower energy consumption. Petri-net-based energy modelling techniques for embedded systems are

```
1  int main() {
2    while( true ) {
3      loop();
4    }
5    return 0;
6  }
7  void loop(){
8    if((SensorLeft.read()==LOW) &&
9      (SensorRight.read()==LOW)) {
10     MoveForward();
11   }
12   if((SensorLeft.read()==HIGH) &&
13     (SensorRight.read()==HIGH)) {
14     Stop();
15   }
16   if((SensorLeft.read()==LOW) &&
17     (SensorRight.read()==HIGH)) {
18     TurnLeft();
19   }
20   if((SensorLeft.read()==HIGH) &&
21     (SensorRight.read()==LOW)) {
22     TurnRight();
23   }
24 }
25 void MoveForward() {
26   MotorLeft.Forward();
27   MotorRight.Forward();
28   delay(20);
29 }
```

Figure 4.13: Excerpt of line follower program

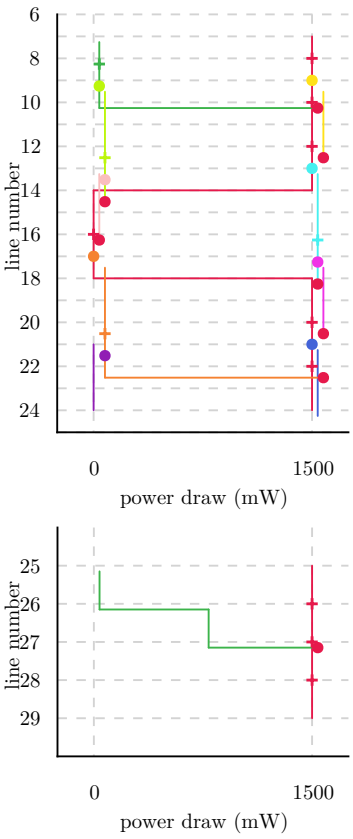


Figure 4.14: Diagrams for *loop* (top) and *MoveForward* (bottom)

proposed by Junior et al. [2006]; Nogueira et al. [2011].

A general analysis for resource consumption is described by Kersten et al. [2012]. There are generic resource consumption analyses, built on techniques such as solving recurrence relations [Albert et al., 2008], amortized analysis [Hoffmann et al., 2011], separation logic [Atkey, 2010], and a Vienna Development Method style program logic [Aspinall et al., 2007]. Contrary to these approaches, our method has an explicit hardware model and a context in the form of component states. This enables the inclusion of state-dependent energy consumption.

Jayaseelan et al. [2006] and Kerrison et al. [2013] analyse energy consumption of the processor running embedded software for specific architectures (SimpleScalar and XMOS ISA-level models, respectively), while our approach is hardware-parametric and focuses on external hardware. Several tools perform a static analysis of the energy consumption of the CPU based on per-instruction measurements, such as in [Sinha and Chandrakasan, 2001; Brooks et al., 2000].

4.8 Discussion and Future Work

This article proposes a new approach for visualizing the energy consumption of a system with external hardware without actually running the software or having a real test setup. The result is presented as skyline diagrams with a direct link to the source code, using line numbers. This visualization is generated by symbolic execution, followed by a merging algorithm to deal with the explosion of possible execution paths. There are few restrictions on the models of hardware components, allowing a user to model a wide variety of hardware components.

We have implemented all techniques of this article in Haskell as a proof of concept, using the Z3 SMT library to prune infeasible paths. Every skyline diagram in this paper and on the project website [Klinik et al., a] was computed by this tool in a few seconds on a ten year old laptop. However, since symbolic execution has exponential complexity, it would only take a couple of nested loops containing component calls for the analysis to no longer be computed in feasible time.

The focus of this paper is on a minimal implementation, to explore the idea of drawing graphs over source lines. A larger case study using the implementation could result in useful feedback on how the process can be applied in practice. This case study should evaluate if programmers get feedback they can use, and if there is a practical need to use another technique instead of, or alongside, bounded symbolic execution. In particular, our visualization should lend itself well to abstract interpretation, which can be an alternative to symbolic execution. This would require incorporating parts of the merging algorithm into the abstract interpretation. A combination of symbolic execution and abstract interpretation would be more complex, but could provide powerful tooling.

Editor integration can improve usability, for instance by annotating or overlaying the source code with diagrams. It may also be useful to offer an interactive visualization that allows developers to explore skylines and recover information about individual execution paths, highlighting the relevant code.

Finally, our approach could track other resources. A similar methodology could be used to visualize memory usage, or even time. *Incidental*, one time, energy consumption of hardware component calls could also be relevant to show.

4.9 Complete Semantics

This section did not make it into the published paper, and was originally made available online as an informal techreport. In section 4.4 we described an algorithm for symbolic execution. In this section, we formally define the semantics in a form common to programming language research.

SECA comes with four semantics, the *standard semantics*, the *standard energy-aware semantics*, the *symbolic semantics* and the *symbolic energy-aware semantics*. Each of them is defined by three evaluation relations: *execute* for whole programs, *step* for statements, and *evaluate* for expressions. Table 4.1 shows an overview.

semantics	programs *execute*	statements *step*	expressions *evaluate*
standard	$\twoheadrightarrow$	$\mapsto$	$\downarrow$
standard energy-aware	$\hat{\twoheadrightarrow}$	$\hat{\mapsto}$	$\hat{\downarrow}$
symbolic	$\rightrightarrows$	$\Mapsto$	$\Downarrow$
symbolic energy-aware	$\hat{\rightrightarrows}$	$\hat{\Mapsto}$	$\hat{\Downarrow}$

Table 4.1: The four semantics of SECA, each consisting of three evaluation relations

4.9.1 Preliminary Definitions

Component calls can perform I/O, which all semantics must anticipate. The presentation in section 4.3.2 uses the IO monad for this reason. For the formal presentation of the semantics in this section, it is more useful to use the simplified view that IO actions pass around a unique pointer to the real world. The transition function of component models (definition 4.9.1) has type $L \times S \times \mathit{World} \to \mathbb{Z} \times S \times \mathit{World}$. This presentation lends itself better to formal treatment than the Haskell equivalent $L \times S \to \mathit{IO}(\mathbb{Z} \times S)$, which is better for implementation.

Definition 4.9.1. (Hardware Component Model) A hardware component model consists of a set of states S, a finite set of labels L, a state transition function $\delta : L \times S \times \mathit{World} \to \mathbb{Z} \times S \times \mathit{World}$, and a power draw function $o : S \to \mathbb{N}$.

The scoping rules for local and global variables are implemented by the functions *assign* and *lookup*. They take a pair of PStates, one for global and one for local variables, use them in the manner defined as follows.

Definition 4.9.2. (Scoping rules) The function $\mathit{assign} : \mathit{Var} \times \mathit{Val} \times \mathit{PState}^2 \to \mathit{PState}^2$ implements the scoping rules of SECA. It takes a variable name x, a value v and a local and a global PState, and assigns the variable according to the following rules.

- If a local variable named x exists, the local PState is updated with v,
- else if a global variable named x exists, the global PState is updated with v,
- else the local PState is extended with a new variable x with value v.

The function $\mathit{lookup} : \mathit{Var} \times \mathit{PState}^2 \to \mathit{Val}$ retrieves values according to the scoping rules. It first looks in the local PState, and if there is no variable named x, it looks in the global PState. We assume that x exists in at least one of the PStates.

When working with lists, $[\,]$ denotes the empty list and $(s : \overline{ss})$ denotes the list with at least one element s and rest $\overline{ss}$. Concatenation of lists is denoted with $(++)$.

Definition 4.9.3. The function $\mathit{curDraw} : \mathit{CState} \to \mathbb{N}$ calculates the total current power draw by summing up the current power draw of all components in the given CState.

4.9.2 The Standard Semantics

We first describe the standard semantics, which is purely concerned with reducing terms to their normal forms. This semantics executes component calls only by updating component states, but ignores power draw.

The standard *step* semantics ($\mapsto$) is given as a small-step structural operational semantics, defined as the smallest relation closed under the rules in fig. 4.15. The rules are explained in detail below. The rules have judgements of the following form.

$$Stmt, PState^2, CState, World, \overline{Stmt} \mapsto PState^2, CState, World, \overline{Stmt}$$

The first *Stmt* in the relation is a statement as defined in fig. 4.3. It is the current statement to be executed. The two PStates are program states, one for global variables and one for local variables. Local variables have function scope, and include the function's actual parameters and variables created anywhere in the function. The CState is the component state, keeping track of all present hardware components. The World is needed because component calls can ask for user input, and perform other kinds of I/O. Only the rule **[e-comp-call]** makes use of it, all other rules just pass the World on unchanged. Finally, there is a list of statements $\overline{Stmt}$, which acts as the program counter. The program counter contains all statements to be executed after the current one.

The semantics essentially implements depth-first traversal of abstract syntax trees, where the program counter is a stack representing the traversal state. We track the traversal state explicitly, so that control flow operations like *while* and *return* can modify it. The rule names in fig. 4.15 are prefixed with an *s*, which stands for *step*.

[s-assign] The statement `x = e` evaluates expression *e* in the current $PState^2$ *pst* to a value *v* and updates *pst* according to the scoping rules.

[s-if-true], **[s-if-false]** The rules for conditionals first evaluate the condition expression *e*, and depending on the outcome prepend either the then-branch or the else-branch to the program counter.

[s-while-true] If the condition *e* evaluates to `true`, this rule prepends the loop body $\overline{s_1}$ and the loop itself to the program counter. It also increments the iteration count *i*.

[s-while-false] If the condition evaluates to `false`, this rule leaves the program counter unchanged, causing the loop to be skipped and the rest of the program to be executed.

[s-return] This rule has two effects. First, it evaluates *e* and assigns the result *v* to a special variable named #*return* in the current local PState. The calling function can use #*return* to access the returned value if required. Second, the return statement discards the program counter and returns the empty list. This stops execution of the function.

[s-expr] This rule lifts evaluation of expressions into execution of statements. The result value *v* is discarded, only the result states are relevant.

The semantics for expressions, called *evaluate* ($\downarrow$), is a big-step semantics, defined by the rules in fig. 4.16. It has judgements of the following form.

$$Expr, PState^2, CState, World \downarrow Val, PState^2, CState, World$$

Expr is the current expression to be evaluated. PStates, CState, and World are needed for function calls. Most rules of ($\downarrow$) are pretty straight-forward, so we discuss only a couple of them here. The rule names are prefixed with an *e*, which stands for *evaluate*.

[e-const] As usual for big-step semantics, there are axioms for constants. In our case *v* are numbers and Boolean values.

$$\textbf{[s-assign]}\ \frac{e, pst, cst, w \downarrow v, pst', cst', w'}{\mathsf{x = e}, pst, cst, w, \overline{pc} \mapsto assign(x, v, pst'), cst', w', \overline{pc}}$$

$$\textbf{[s-if-true]}\ \frac{e, pst, cst, w \downarrow \mathsf{true}, pst', cst', w'}{\textbf{if}(e)\ \{\ \overline{s_1}\ \}\ \textbf{else}\ \{\ \overline{s_2}\ \}, pst, cst, w, \overline{pc} \mapsto pst', cst', w', \overline{s_1} +\!+ \overline{pc}}$$

$$\textbf{[s-if-false]}\ \frac{e, pst, cst, w \downarrow \mathsf{false}, pst', cst', w'}{\textbf{if}(e)\ \{\ \overline{s_1}\ \}\ \textbf{else}\ \{\ \overline{s_2}\ \}, pst, cst, w, \overline{pc} \mapsto pst', cst', w', \overline{s_2} +\!+ \overline{pc}}$$

$$\textbf{[s-while-true]}\ \frac{e, pst, cst, w \downarrow \mathsf{true}, pst', cst', w'}{s, pst, cst, w, \overline{pc} \mapsto pst', cst', w', (\overline{s_1} +\!+ (s' : \overline{pc}))}$$

where
$s = \textbf{while}(e)\ \{\ \overline{s_1}\ \}\ i$
$s' = \textbf{while}(e)\ \{\ \overline{s_1}\ \}\ (i + 1)$

$$\textbf{[s-while-false]}\ \frac{e, pst, cst, w \downarrow \mathsf{false}, pst', cst', w'}{\textbf{while}(e)\ \{\ \overline{s}\ \}\ i, pst, cst, w, \overline{pc} \mapsto pst', cst', w', \overline{pc}}$$

$$\textbf{[s-return]}\ \frac{e, pst, cst, w \downarrow v, pst', cst', w'}{\textbf{return}\ e, pst, cst, w, \overline{pc} \mapsto assign(\#return, v, pst'), cst', w', []}$$

$$\textbf{[s-expr]}\ \frac{e, pst, cst, w \downarrow v, pst', cst', w'}{e, pst, cst, w, \overline{pc} \mapsto pst', cst', w', \overline{pc}}$$

Figure 4.15: Small-step standard semantics ($\mapsto$) for statements

[e-var] Variables are evaluated by looking up their value according to the scoping rules, see definition 4.9.2.

[e-add] Arithmetic addition is evaluated by first evaluating the argument expressions and then calculating the result value. Similar rules exist for the other operators.

[e-compare-true] Comparisons are evaluated by evaluating the argument expressions, and yielding true if the left-hand side is less than or equal to the right-hand side. Similar rules exist for when the comparison is false, and for < and ==.

[e-func-call] This rule handles function calls $f(\overline{e})$. It does two things. First, it evaluates all actual parameter expressions $\overline{e}$, in sequence from left to right, while threading the state through these evaluations. This operation is denoted by $(\downarrow^*)$. The result of evaluating the parameters is a list of values $\overline{v}$, a new component state cst', and new local and global PStates lst' and gst'.

Second, it performs the actual function call. We assume that a function definition with name f exists, with formal parameters $\overline{x}$ and body $\overline{s}$. Let fst be a new local PState for f, where each variable x is initialized with the corresponding actual argument value v. The body $\overline{s}$ is then executed in fst, with global CState cst' and global PState gst'. This execution ends when all statements of $\overline{s}$ have been processed, resulting in a new CState cst'' and PStates fst' and gst''. The resulting local state fst' is discarded.

The rule finally returns the global CState cst'' and PState gst'' resulting from the function call, and the local PState lst' and program counter $\overline{pc}$ from before the function call.

When execution of the function body terminates, we know that the return register *#return* in the function's local PState holds the function's return value.

[e-comp-call] A component call c.f() causes component c to make an f-transition. The current state of c is s_c. The transition $\delta_c(f, s_c, w)$ results in a return value r, a new state s'_c, and an updated world w'. The new CState cst' is cst where c is now in state s'_c. We assume that the semantics knows what kind of component c is to use the right δ_c.

Execution of statement lists $(\twoheadrightarrow)$ is defined as the repeated application of $(\mapsto)$ until the program counter is empty. When the program counter is empty, $(\twoheadrightarrow)$ can no longer make a step, and execution of the statements is done. Formally: $(\twoheadrightarrow)$ is the smallest relation closed under the rules in fig. 4.17. It has judgements of the following form.

$$PState^2, CState, World, \overline{Stmt} \twoheadrightarrow PState^2, CState, World, \overline{Stmt}$$

Execution of complete programs combines all relations seen so far: $(\mapsto)$, $(\downarrow)$, and $(\twoheadrightarrow)$. To execute a complete SECA program, we need the following ingredients.

- A list of function definitions, one of them int main().
- A list of global variable definitions.
- An initial CState containing all components mentioned in the program, beginning in their start state.

The program is executed by first initializing the global variables, and then running the body of *main* in this global PState, an empty local PState, and the initial CState. Global variables can only be initialized with constant expressions, which may refer only to global variables defined earlier. There exists a simplified expression evaluator for initializers of global variables that does not support function calls. This evaluator is not further discussed here.

$$\textbf{[e-const]}\ \frac{}{v, pst, cst, w \downarrow v, pst, cst, w}$$

$$\textbf{[e-var]}\ \frac{lookup(x, pst) = v}{x, pst, cst, w \downarrow v, pst, cst, w}$$

$$\textbf{[e-add]}\ \frac{\begin{array}{c} e_1, pst, cst, w \downarrow i_1, pst', cst', w' \\ e_2, pst', cst', w' \downarrow i_2, pst'', cst'', w'' \\ v = i_1 + i_2 \end{array}}{e_1 + e_1, pst, cst, w \downarrow v, pst'', cst'', w''}$$

$$\textbf{[e-compare-true]}\ \frac{\begin{array}{c} e_1, pst, cst, w \downarrow i_1, pst', cst', w' \\ e_2, pst', cst', w' \downarrow i_2, pst'', cst'', w'' \\ i_1 \leq i_2 \end{array}}{e_1 \leq e_1, pst, cst, w \downarrow \mathsf{true}, pst'', cst'', w''}$$

$$\textbf{[e-func-call]}\ \frac{\begin{array}{c} \overline{e}, pst, cst, w \downarrow^* \overline{v}, \langle lst', gst' \rangle, cst', w' \\ \langle fst, gst' \rangle, cst', w', \overline{s} \twoheadrightarrow \langle fst', gst'' \rangle, cst'', w'', [\,] \\ lookup(\#return, fst') = r \end{array}}{\begin{array}{l} f(\overline{e}), pst, cst, w \downarrow r, \langle lst', gst'' \rangle, cst'', w'' \\ \text{where} \\ f(\overline{x})\{\ \overline{s}\ \} \text{ is a defined function} \\ fst = [\overline{x} \mapsto \overline{v}] \text{ is the initial local state for } f \end{array}}$$

$$\textbf{[e-comp-call]}\ \frac{\delta_c(f, s_c, w) = \langle r, s_c', w' \rangle \qquad cst' = cst[c \mapsto s_c']}{c.f(), pst, cst, w \downarrow r, pst, cst', w'}$$

Figure 4.16: Excerpt of the big-step standard semantics (↓) for expressions

$$\frac{}{pst, cst, w, [\,] \twoheadrightarrow pst, cst, w, [\,]}$$

$$\frac{\begin{array}{c} s, pst, cst, w, \overline{pc} \mapsto pst', cst', w', \overline{pc}' \\ pst', cst', w', \overline{pc}' \twoheadrightarrow pst'', cst'', w'', \overline{pc}'' \end{array}}{pst, cst, w, (s : \overline{pc}) \twoheadrightarrow pst'', cst'', w'', \overline{pc}''}$$

Figure 4.17: Big-step standard semantics (↠) for statement lists

4.9.3 The Energy-Aware Standard Semantics

The energy-aware semantics of SECA is an extension of the standard semantics. It has similar reduction rules, but additionally keeps track of the energy consumption of each function, represented by a skyline over program points. For each of the standard semantics ($\mapsto$, $\twoheadrightarrow$, $\downarrow$) there exists an energy-aware extension ($\hat{\mapsto}$, $\hat{\twoheadrightarrow}$, $\hat{\downarrow}$).

The energy-aware semantics for statements ($\hat{\mapsto}$) assembles skylines as execution proceeds. The rules keep track of the current skyline, and every rule adds one or more segments. Every function call starts a new current skyline for that function, and every return statement adds the current skyline to a register of the function's skylines. Skylines are lists of skyline segments, as defined in section 4.3.2. The energy-aware semantics for statements ($\hat{\mapsto}$) has judgements of the following form.

$$\begin{array}{rl} & \mathit{Stmt}, \mathit{PState}^2, \mathit{CState}, \mathit{World}, \overline{\mathit{Stmt}}, \mathit{Skyline}, \mathit{Skylines} \\ \hat{\mapsto} & \mathit{PState}^2, \mathit{CState}, \mathit{World}, \overline{\mathit{Stmt}}, \mathit{Skyline}, \mathit{Skylines} \end{array}$$

$\mathit{PState}^2, \mathit{CState}, \mathit{World}, \overline{\mathit{Stmt}}$ are as in ($\mapsto$). *Skyline* is the so-far assembled skyline of the currently executed function. *Skylines* is a finite mapping from function names to lists of skylines. It registers for every function the skylines that result from all calls to that function. The rules of $\hat{\mapsto}$ are shown in fig. 4.18. For statements s, we write s_l to indicate that s occurs on line l in the source file. For example, $\mathsf{x}_l = e$ is an assignment statement to variable x where the variable name occurs on line l.

The rules of ($\hat{\mapsto}$) have the same reduction behaviour for programs as the rules for ($\mapsto$). In addition, they construct skylines that represent the energy behaviour of the program execution. Skylines are extended in reverse order for notational convenience. The rule names are prefixed with *es*, which stands for *energy-aware step*.

[es-assign] This rule first extends the skyline with a forwards line $F(l)$ to the line of the variable name x of the assignment statement. This extended skyline is passed to the evaluation of e. The order is important, because e itself can have side effects and extend the skyline. The pattern of first extending the skyline and then evaluating subexpressions is used in the other rules as well.

[es-if-true], [es-if-false] Conditionals extend the skyline to the line of the **if** keyword, before evaluating the condition e. The rule **[es-if-false]** does the same, and is not shown.

[es-while] The rule for while loops is the most complicated one. We have collapsed the cases when the condition is true and false in one rule, because the energy behaviour is the same. The only difference is the resulting continuation, which is identical to **[s-while-true]** and **[s-while-false]**.

This rule makes use of the loop counter i, because the generated skyline must be different if the loop is encountered for the first time. If $i = 0$ the loop is executed for the first time, and the skyline *sky* comes from outside the loop. In this case *sky* is extended with a forwards line to the location l of the **while** keyword. If $i > 0$ the loop has been executed before, and *sky* comes from inside the loop body. In this case *sky* is extended with a forwards line $F(m)$ to the end of the loop, and then jumps $J(l)$ back to the beginning of the loop. The extended skyline *startSky* is then used for evaluation of the condition e.

[es-return] The rule for return statements first extends the current skyline *sky* with a forwards line $F(l)$ to the location of the **return** keyword, and then evaluates e.

$$\textbf{[es-assign]}\ \frac{e, pst, cst, w, sky +\!\!+ [F(l)], skies \hat{\downarrow} v, pst', cst', w', sky', skies'}{\mathsf{x}_l = \mathsf{e}, pst, cst, w, \overline{pc}, sky, skies \hat{\mapsto} assign(\mathsf{x}, v, pst'), cst', w', sky', skies'}$$

$$\textbf{[es-if-true]}\ \frac{e, pst, cst, w, sky +\!\!+ [F(l)], skies \hat{\downarrow} \mathsf{true}, pst', cst', w', sky', skies'}{\begin{array}{l}\textbf{if}_l(e)\ \{\ \overline{s_1}\ \}\ \textbf{else}\ \{\ \overline{s_2}\ \}, pst, cst, w, sky, skies \\ \hat{\mapsto} pst', cst', w', (\overline{s_1} +\!\!+ \overline{pc}), sky', skies'\end{array}}$$

$$\textbf{[es-while]}\ \frac{e, pst, cst, w, startSky, skies \hat{\downarrow} v, pst', cst', w', sky', skies'}{s, pst, cst, w, \overline{pc}, sky, skies \hat{\mapsto} cst', pst', w', \overline{pc}', sky', skies'}$$

where

$$s = \textbf{while}_l(e)\ \{\ \overline{s_1}\ \}_m\ i$$

$$startSky = \begin{cases} sky +\!\!+ [F(l)] & \text{if } i = 0 \\ sky +\!\!+ [F(m), J(l)] & \text{if } i > 0 \end{cases}$$

$$\overline{pc}' = \begin{cases} \overline{s_1} +\!\!+ [\textbf{while}_l(e)\ \{\ \overline{s_1}\ \}_m\ (i+1)] +\!\!+ \overline{pc} & \text{if } v = \mathsf{true} \\ \overline{pc} & \text{if } v = \mathsf{false} \end{cases}$$

$$\textbf{[es-return]}\ \frac{e, pst, cst, w, sky +\!\!+ [F(l)], skies \hat{\downarrow} v, pst', cst', w', sky', skies'}{\begin{array}{l}\textbf{return}_l\ e, pst, cst, w, \overline{pc}, sky, skies \\ \hat{\mapsto} assign(\#return, v, pst'), cst', w', [\,], sky', skies'\end{array}}$$

$$\textbf{[es-expr]}\ \frac{e, pst, cst, w, sky, skies \hat{\downarrow} v, pst', cst', w', sky', skies'}{e, pst, cst, w, \overline{pc}, sky, skies \hat{\mapsto} pst', cst', w', \overline{pc}, sky', skies'}$$

Figure 4.18: Energy-aware small-step semantics ($\hat{\mapsto}$) for statements

[es-expr] This rule lifts evaluation of expressions into the semantics for statements. It just evaluates the expression for its side effect, and discards the returned value.

The energy-aware semantics for evaluating expressions ($\hat{\downarrow}$) is a big-step semantics, based on ($\downarrow$). It has judgements of the form

$$Expr, PState^2, CState, World, Skyline, Skylines$$
$$\hat{\downarrow} Val,\ \ PState^2, CState, World, Skyline, Skylines$$

where $PState^2, CState, World$ are as for ($\downarrow$). The semantics is defined by the rules in fig. 4.19. Most of the rules are the same as in ($\downarrow$), and just additionally pass on the skylines unchanged. We omit those rules. The only interesting rules are **[ee-func-call]** and **[ee-comp-call]**. The rule names are prefixed with *ee*, which stands for *energy-aware evaluate*.

[ee-func-call] A function call first evaluates all actual parameters $\overline{e}$ with the state-chaining extension ($\hat{\downarrow}^*$) of ($\hat{\downarrow}$). This results in a skyline sky'. The rule then executes the function body using a new skyline initialized with a start segment $[S(m, d')]$ where m is the line

$$\textbf{[ee-func-call]}\ \frac{\begin{array}{c}\overline{e}, pst, cst, w, sky, skies \mathrel{\hat{\downarrow}^*} \overline{v}, \langle lst', gst'\rangle, cst', w', sky', skies' \\ \langle fst, gst'\rangle, cst', w', \overline{s}, [S(m,d')], skies' \mathrel{\hat{\twoheadrightarrow}} \langle fst', gst''\rangle, cst'', w'', [], fsky, skies'' \\ lookup(\#return, fst') = r\end{array}}{\begin{array}{l} f(\overline{e})_l, pst, cst, w, sky, skies \mathrel{\hat{\downarrow}} r, \langle lst', gst''\rangle, cst'', w'', sky'', skies''' \\ \text{where} \\ f(\overline{x})\{_m\ \overline{s}\ \}_n \text{ is a defined function} \\ fst = [\overline{x} \mapsto \overline{v}] \text{ is the initial local state for } f \\ d' = curDraw(c') \text{ is the power draw after evaluating the arguments} \\ d'' = curDraw(c'') \text{ is the power draw after executing } f \\ sky'' = sky' \mathbin{++} [F(l), E(d'')] \\ skies''' = skies''[f \mapsto skies''(f) \cup \{fsky \mathbin{++} [F(n)]\}] \end{array}}$$

$$\textbf{[ee-comp-call]}\ \frac{\delta_c(f, s_c, w) = \langle r, s_c', w'\rangle \qquad cst' = cst[c \mapsto s_c'] \qquad d = curDraw(cst')}{c.f()_l, pst, cst, w, sky, skies \mathrel{\hat{\downarrow}} r, pst, cst', w', sky \mathbin{++} [F(l), E(d)], skies}$$

Figure 4.19: Excerpt of the energy-aware expression evaluation semantics ($\hat{\downarrow}$)

where f starts, and d' is the total power draw of all components after evaluation of the actual parameters. When the call to f returns with skyline $fsky$, we know that $fsky$ ends at the return statement that caused f to return. $fsky$ is extended with a forwards segment $F(n)$ to the end of f. This skyline is then added to the register of the skylines of all calls to f.

The calling function finally extends its own skyline with an edge $[F(l), E(d'')]$ where l is the location of the call site, and d'' the total power draw after the call to f.

[ee-comp-call] Component calls c.f() generate edges in the following way. First, the f-transition of the current state of c is calculated, resulting in a return value r, a new state s_c', and a new world w'. Second, The CState is updated so that c is now in state s. Third, The power draw d is calculated in the new CState cst'. Finally the skyline is extended with an edge $[F(l), E(d)]$ at the location l of the component call, to power draw d.

The energy-aware execution semantics ($\hat{\twoheadrightarrow}$) is defined similarly to ($\twoheadrightarrow$), and not shown here.

4.9.4 The Symbolic Execution Semantics

Our final goal, the energy-aware symbolic execution semantics, combines symbolic execution with energy consumption analysis. The result is a skyline for each path through a program, together with the path constraint that leads to the skyline. We present the energy-aware symbolic execution semantics in two steps, similar to the presentation of the standard semantics. First we define the symbolic execution semantics for pure programs, and then extend it with energy consumption analysis. For this we need to define six more relations, three for symbolic execution, and three for energy-aware symbolic execution. There is *symbolic step* ($\mapsto$) for symbolic execution of statements, *symbolic evaluation* ($\Downarrow$) for expressions, and *symbolic execution* ($\rightrightarrows$) for whole programs. We are using double

arrows to hint that one program can have multiple symbolic executions. Their energy-aware counterparts are *energy-aware symbolic step* ($\hat{\Mapsto}$), *energy-aware symbolic evaluation* ($\hat{\Downarrow}$), and *energy-aware symbolic execution* ($\hat{\rightrightarrows}$).

Symbolic execution of statements ($\mapsto$) is a relation with judgements of the form

$$Stmt, SPState^2, SCState, \overline{Stmt}, SVal \mapsto SPState^2, SCState, \overline{Stmt}, SVal$$

where the first Stmt is the statement to be executed, $SPState^2$ are program states for variables holding symbolic values, *SCState* is a symbolic component state, $\overline{Stmt}$ the program counter, and *SVal* the path constraint.

The relation is defined by the rules in fig. 4.20. The rule names are prefixed with *ss*, which stands for *symbolic step*. This relation is not right-unique, which means that the same left-hand side can be related to more than one right-hand side. This is the essence of symbolic execution, where one program can have multiple executions. The set of successor states of a statement *s* can be recovered by finding all right-hand sides that *s* relates to. This process can be repeated to recover the tree of possible execution paths of a program.

[ss-assign] Evaluates the symbolic expression *se* to a symbolic value *sv*, and updates the SPState according to the scoping rules. Note that, because expression evaluation is also not right-unique, whenever an expression can evaluate to multiple values, this rule generates just as many successor states. The same holds for all rules that evaluate expressions.

[ss-if-true], **[ss-if-false]** These rules are the heart of symbolic execution. Both rules always match any conditional, and they produce successor states for the then-branch and the else-branch respectively. The path constraint in **[ss-if-true]** is extended with the condition for the then-branch, and in **[ss-if-false]** with the negated condition for the else-branch.

[ss-while-true], **[ss-while-false]** Similar to the rules for conditionals, both rules always match, producing successor states for skipping the loop with negated condition, and looping with positive condition. Note that **[ss-while-true]** always appends the while-loop itself to the program counter, thereby creating infinite execution paths for any program that contains a while-loop. This is not a problem for the mathematical definition of the semantics. An implementation however must take precautions to not run indefinitely. Our implementation takes two measures for this. First, it employs an SMT solver to prune infeasible paths, which automatically terminates loops if the condition can no longer be satisfied. Second, it limits the number of times a loop is executed, and forcibly cuts execution when the limit is reached. This guarantees execution to terminate, by whichever measure comes into action first.

[ss-return] This rule works as in the standard semantics, emptying the program counter $\overline{pc}$, and updating the #*return* register in the program state.

[ss-expr] This rule, as in the standard semantics, evaluates the expression for its side effects and discards the result value.

The symbolic execution semantics for evaluating expressions ($\Downarrow$) is defined by the rules in fig. 4.21. The semantics has judgements of the following form.

$$SExpr, SPState^2, SCState, SVal \Downarrow SVal, SPState^2, SCState, SVal$$

$$\textbf{[ss-assign]}\ \frac{se, pst, cst, \varphi \Downarrow sv, pst', cst', \varphi'}{\mathsf{x} = se, pst, cst, \overline{pc}, \varphi \Longmapsto assign(x, sv, pst'), cst', \overline{pc}, \varphi'}$$

$$\textbf{[ss-if-true]}\ \frac{se, pst, cst, \varphi \Downarrow sv, pst', cst', \varphi'}{\textbf{if}(se)\ \{\ \overline{s_1}\ \}\ \textbf{else}\ \{\ \overline{s_2}\ \}, pst, cst, \overline{pc}, \varphi \Longmapsto pst', cst', (\overline{s_1} +\!\!+ \overline{pc}), \varphi' \wedge sv}$$

$$\textbf{[ss-if-false]}\ \frac{se, pst, cst, \varphi \Downarrow sv, pst', cst', \varphi'}{\textbf{if}(se)\ \{\ \overline{s_1}\ \}\ \textbf{else}\ \{\ \overline{s_2}\ \}, pst, cst, \overline{pc}, \varphi \Longmapsto pst', cst', (\overline{s_2} +\!\!+ \overline{pc}), \varphi' \wedge \neg sv}$$

$$\textbf{[ss-while-true]}\ \frac{se, pst, cst, \varphi \Downarrow sv, pst', cst', \varphi'}{\textbf{while}(se)\ \{\ \overline{s}\ \}\ i, pst, cst, \overline{pc}, \varphi \Longmapsto pst', cst', \overline{pc}', \varphi' \wedge sv}$$

where

$$\overline{pc}' = \overline{s} +\!\!+ [\textbf{while}(se)\ \{\ \overline{s}\ \}\ (i+1)] +\!\!+ \overline{pc}$$

$$\textbf{[ss-while-false]}\ \frac{se, pst, cst, \varphi \Downarrow sv, pst', cst', \varphi'}{\textbf{while}(se)\ \{\ \overline{s}\ \}\ i, pst, cst, \overline{pc}, \varphi \Longmapsto pst', cst', \overline{pc}, \varphi' \wedge \neg sv}$$

$$\textbf{[ss-return]}\ \frac{se, pst, cst, \varphi \Downarrow sv, pst', cst', \varphi'}{\textbf{return}\ se, pst, cst, \overline{pc}, \varphi \Longmapsto assign(\#return, sv, pst'), cst', [\,], \varphi'}$$

$$\textbf{[ss-expr]}\ \frac{se, pst, cst, \varphi \Downarrow sv, pst', cst', \varphi'}{se, pst, cst, \overline{pc}, \varphi \Longmapsto pst', cst', \overline{pc}, \varphi'}$$

Figure 4.20: Symbolic small-step semantics ($\Longmapsto$) for statements

$$\textbf{[se-const]}\ \frac{}{sv, pst, cst, \varphi \Downarrow sv, pst, cst, \varphi}$$

$$\textbf{[se-var]}\ \frac{lookup(x, pst) = sv}{x, pst, cst, \varphi \Downarrow sv, pst, cst, \varphi}$$

$$\textbf{[se-add]}\ \frac{se_1, pst, cst, \varphi \Downarrow sv_1, pst', cst', \varphi' \qquad se_2, pst', cst', \varphi' \Downarrow sv_2, pst'', cst'', \varphi''}{se_1 + se_2, pst, cst, \varphi \Downarrow sv_1 + sv_2, pst'', cst'', \varphi''}$$

$$\textbf{[se-func-call]}\ \frac{\begin{array}{c}\overline{se}, pst, cst, \varphi \Downarrow^* \overline{sv}, \langle lst', gst' \rangle, cst', \varphi' \\ \langle fst, gst' \rangle, cst', \varphi', \overline{s} \rightrightarrows \langle fst', gst'' \rangle, cst'', \varphi'', [\,] \\ lookup(\#return, fst') = r\end{array}}{\begin{array}{l}f(\overline{se}), pst, cst, \varphi \Downarrow r, \langle lst', gst'' \rangle, cst'', \varphi'' \\ \text{where} \\ f(\overline{x})\{\ \overline{s}\ \} \text{ is a defined function} \\ fst = [\overline{x} \mapsto \overline{v}] \text{ is the initial local state for } f\end{array}}$$

$$\textbf{[se-comp-call]}\ \frac{\delta_c(f, s_c) = \langle sv, \varphi', s_c' \rangle \qquad cst' = cst[c \mapsto s_c']}{c.f(), pst, cst, \varphi \Downarrow sv, pst, cst', \varphi \wedge \varphi'}$$

Figure 4.21: Excerpt of the symbolic expression evaluation semantics ($\Downarrow$)

The semantics relates symbolic expressions *SExpr* to symbolic values *SVal* in the context of a symbolic program state *SPState*[2], a symbolic component state *SCState*, and a path constraint *SVal*. The rule names are prefixed with *se*, which stands for *symbolic evaluate*.

[se-const] Constants evaluate to themselves, just like in the standard semantics.

[se-var] Variables are evaluated by looking up their value in the program state.

[se-add] Adding two symbolic expressions se_1 and se_2 requires evaluating them to symbolic values sv_1 and sv_2 first. As with all symbolic rules, both premises can match multiple times, in which case the conclusion is applied accordingly. The result is not a single number, but the compound symbolic value $sv_1 + sv_2$.

[se-func-call] Symbolic function calls work just like standard function calls, except that all premises can match multiple times, each resulting in a function call with different arguments. Evaluating the arguments $\overline{se}$ requires sequential evaluation of each argument while threading the state through all these evaluations. This operation is denoted by $\Downarrow^*$.

[se-comp-call] Symbolic component calls yield a symbolic value *sv*, possibly containing symbolic inputs, together with a boolean predicate φ' over those symbolic inputs. The component call evaluates to *sv*, and the path constraint φ is extended with φ'.

The symbolic execution semantics ($\rightrightarrows$) for statement lists is defined similarly to ($\twoheadrightarrow$), and is not shown here.

4.9.5 The Energy-Aware Symbolic Execution Semantics

We are finally ready to define the energy-aware symbolic execution semantics for SECA. It is a combination of the symbolic execution semantics and the energy-aware standard semantics. It tracks path constraints and energy skylines for each execution path. The result is a set of energy skylines, together with their path constraint.

Energy-aware symbolic execution of statement lists ($\hat{\rightrightarrows}$) is the relation which executes statements of the program counter $\overline{s}$ stepwise until it is empty. It has the following form.

$$\begin{aligned} &\mathit{SPState}^2, \mathit{SCState}, \overline{\mathit{Stmt}}, \mathit{SVal}, \mathit{Skyline}, \mathit{Skylines} \\ \hat{\rightrightarrows}\ &\mathit{SPState}^2, \mathit{SCState}, \overline{\mathit{Stmt}}, \mathit{SVal}, \mathit{Skyline}, \mathit{Skylines} \end{aligned}$$

The relation is defined analogous to ($\twoheadrightarrow$), and not shown here.

Energy-aware symbolic stepping ($\hat{\Mapsto}$) executes single statements while tracking their energy skylines. It is a relation with judgements of the following form.

$$\begin{aligned} &\mathit{Stmt}, \mathit{SPState}^2, \mathit{SCState}, \overline{\mathit{Stmt}}, \mathit{SVal}, \mathit{Skyline}, \mathit{Skylines} \\ \hat{\Mapsto}\quad &\mathit{SPState}^2, \mathit{SCState}, \overline{\mathit{Stmt}}, \mathit{SVal}, \mathit{Skyline}, \mathit{Skylines} \end{aligned}$$

The relation is defined by the rules in fig. 4.22. The rule names are prefixed with *ess*, which stands for *energy-aware symbolic step*.

[ess-assign] Assignments extend the current skyline with a forward segment $F(l)$ to the location l of the variable. The rule then evaluates *se*, and updates the program state accordingly.

[ess-if-true] This rule extends the skyline with a forward segment $F(l)$ to the location l of the **if** keyword, and evaluates the condition to a symbolic value *sv*. Execution continues as if the condition was true with the then-branch $\overline{s_1}$ and the path constraint $\varphi' \wedge sv$.

[ess-if-false] This rule also extends the skyline to the location of the **if** keyword, but execution continues with the else-branch $\overline{s_2}$ and the path constraint $\varphi' \wedge \neg sv$.

[ess-while-true] This rule evaluates the condition to a value *sv*, and enters the loop with the path constraint $\varphi' \wedge sv$. The skyline is extended to the location of the **while** keyword in one of two ways. If the loop counter is 0, we know that *sky* starts outside the loop, so it is extended with a forwards line to the beginning of the loop $F(l)$. Otherwise, *sky* starts inside the loop, and is extended with a forwards line to the end of the loop $F(m)$, followed by a jump to the beginning of the loop $J(l)$.

[ess-while-false] This rule extends the skyline in the same way as **[ess-while-true]**, but continues execution as if the condition was false by adding $\neg sv$ to the path constraint.

[ess-return] This rule extends *sky* to the location l of the **return** keyword, and then evaluates the return value *sv*. It assigns this value to the return register of the current function's SPState pst', and clears the program counter.

[ess-expr] This rule evaluates the expression *se* for its side effects, and discards the result value.

The energy-aware symbolic execution semantics for evaluating expressions ($\hat{\Downarrow}$) is defined by the rules in fig. 4.23. The rule names are prefixed with *ese*, which stands for *energy-aware symbolic evaluate*. The semantics has judgements of the following form.

$$\begin{aligned} &\mathit{SExpr}, \mathit{SPState}^2, \mathit{SCState}, \mathit{SVal}, \mathit{Skyline}, \mathit{Skylines} \\ \hat{\Downarrow}\ &\mathit{SVal},\ \ \mathit{SPState}^2, \mathit{SCState}, \mathit{SVal}, \mathit{Skyline}, \mathit{Skylines} \end{aligned}$$

$$\textbf{[ess-assign]}\ \frac{se, pst, cst, \varphi, sky \mathbin{+\!\!+} [F(l)], skies \Downarrow sv, pst', cst', \varphi', sky', skies'}{\begin{array}{l}\mathsf{x}_l = se, pst, cst, \overline{pc}, \varphi, sky, skies \\ \hat{\Rightarrow} \ assign(x, sv, pst'), cst', \overline{pc}, \varphi', sky', skies'\end{array}}$$

$$\textbf{[ess-if-true]}\ \frac{se, pst, cst, \varphi, sky \mathbin{+\!\!+} [F(l)], skies \Downarrow sv, pst', cst', \varphi', sky', skies'}{\begin{array}{l}\textbf{if}_l(se)\ \{\ \overline{s_1}\ \}\ \textbf{else}\ \{\ \overline{s_2}\ \}, pst, cst, \overline{pc}, \varphi, sky, skies \\ \hat{\Rightarrow} \ pst', cst', (\overline{s_1} \mathbin{+\!\!+} \overline{pc}), \varphi' \wedge sv, sky', skies'\end{array}}$$

$$\textbf{[ess-if-false]}\ \frac{se, pst, cst, \varphi, sky \mathbin{+\!\!+} [F(l)], skies \Downarrow sv, pst', cst', \varphi', sky', skies'}{\begin{array}{l}\textbf{if}_l(se)\ \{\ \overline{s_1}\ \}\ \textbf{else}\ \{\ \overline{s_2}\ \}, pst, cst, \overline{pc}, \varphi, sky, skies \\ \hat{\Rightarrow} \ pst', cst', (\overline{s_2} \mathbin{+\!\!+} \overline{pc}), \varphi' \wedge \neg sv, sky', skies'\end{array}}$$

$$\textbf{[ess-while-true]}\ \frac{se, pst, cst, \varphi, startSky, skies \Downarrow sv, pst', cst', \varphi', sky', skies'}{\begin{array}{l}\textbf{while}_l(se)\ \{\ \overline{s}\ \}_m\ i, pst, cst, \overline{pc}, \varphi, sky, skies \\ \hat{\Rightarrow} \ pst', cst', \overline{pc}', \varphi' \wedge sv, sky', skies' \\ \text{where} \\ \overline{pc}' = \overline{s} \mathbin{+\!\!+} [\textbf{while}(se)\ \{\ \overline{s}\ \}\ (i+1)] \mathbin{+\!\!+} \overline{pc} \\ startSky = \begin{cases} sky \mathbin{+\!\!+} [F(l)] & \text{if } i = 0 \\ sky \mathbin{+\!\!+} [F(m), J(l)] & \text{if } i > 0 \end{cases}\end{array}}$$

$$\textbf{[ess-while-false]}\ \frac{se, pst, cst, \varphi, startSky, skies \Downarrow sv, pst', cst', \varphi', sky', skies'}{\begin{array}{l}\textbf{while}_l(se)\ \{\ \overline{s}\ \}_m\ i, pst, cst, \overline{pc}, \varphi, sky, skies \\ \hat{\Rightarrow} \ pst', cst', \overline{pc}, \varphi' \wedge \neg sv, sky', skies' \\ \text{where} \\ startSky = \begin{cases} sky \mathbin{+\!\!+} [F(l)] & \text{if } i = 0 \\ sky \mathbin{+\!\!+} [F(m), J(l)] & \text{if } i > 0 \end{cases}\end{array}}$$

$$\textbf{[ess-return]}\ \frac{se, pst, cst, \varphi, sky \mathbin{+\!\!+} F(l), skies \Downarrow sv, pst', cst', \varphi', sky', skies'}{\begin{array}{l}\textbf{return}_l\ se, pst, cst, \overline{pc}, \varphi, sky, skies \\ \hat{\Rightarrow} \ assign(\#return, sv, pst'), cst', [\,], \varphi', sky', skies'\end{array}}$$

$$\textbf{[ess-expr]}\ \frac{se, pst, cst, \varphi, sky, skies \Downarrow sv, pst', cst', \varphi', sky', skies'}{se, pst, cst, \overline{pc}, \varphi, sky, skies \hat{\Rightarrow} pst', cst', \overline{pc}, \varphi', sky', skies'}$$

Figure 4.22: Energy-aware symbolic execution semantics ($\hat{\Rightarrow}$) for statements

The semantics evaluates a symbolic expression *SExpr* to a symbolic value *SVal* in the context of a program state $SPState^2$, a symbolic component state *SCState*, a path constraint *SVal*, the current skyline *Skyline*, and the global skyline register *Skylines*.

[ese-const] Constants evaluate to themselves.

[ese-var] Variables are evaluated by a lookup in the program state.

[ese-add] Additions are evaluated by sequentially evaluating both argument expressions to symbolic values sv_1 and sv_2, and returning the compound symbolic value $sv_1 + sv_2$.

[ese-func-call] The rule for function calls first sequentially evaluates all argument expressions $\overline{se}$ using the sequential evaluation operator $\hat{\Downarrow}^*$. It then initializes a new local SPState *fst* and a new skyline $[S(m, d')]$ for the call to *f*. The body $\overline{s}$ of *f* is then executed by $(\hat{\Rightarrow})$, resulting in a skyline *fsky*. We know that *fsky* ends at the location of the return statement that caused *f* to return, so this rule extends it with a forwards line $F(n)$ to the location of the end of *f*. This skyline is added to the set of skylines of all calls to *f*, $skies'''$. Finally, the skyline of the calling function is extended with an edge $[F(l), E(d'')]$ at the call site *l*, where d'' is the power draw after executing *f*.

[ese-comp-call] Component calls are evaluated by calling the transition function of the component on its current state s_c. This results in a symbolic value *sv*, a boolean predicate φ' on the value, and a new component state s'_c. The component call evaluates to the return value and the path constraint $\varphi \wedge \varphi'$, together with the skyline extended with an edge $[F(l), E(d)]$ where *l* is the location of the component call, and *d* is the power draw after the component call.

Acknowledgements.

We would like to thank Rinus Plasmeijer, Olha Shkaravska, Tim Steenvoorden, and Nico Naus for many hours of fruitful discussion. Special thanks goes to Ralf Hinze, who created an exam question, the grading of which eventually led to the idea of resource skylines. Thanks also to Pieter Koopman who provided funding for this project.

$$\textbf{[ese-const]}\ \frac{}{sv, pst, cst, \varphi, sky, skies \ \hat{\Downarrow}\ sv, pst, cst, \varphi, sky, skies}$$

$$\textbf{[ese-var]}\ \frac{lookup(x, pst) = sv}{x, pst, cst, \varphi, sky, skies \ \hat{\Downarrow}\ sv, pst, cst, \varphi, sky, skies}$$

$$\textbf{[ese-add]}\ \frac{\begin{array}{c} se_1, pst, cst, \varphi, sky, skies \ \hat{\Downarrow}\ sv_1, pst', cst', \varphi', sky', skies' \\ se_2, pst', cst', \varphi', sky', skies' \ \hat{\Downarrow}\ sv_2, pst'', cst'', \varphi'', sky'', skies'' \end{array}}{se_1 + se_2, pst, cst, \varphi, sky, skies \ \hat{\Downarrow}\ sv_1 + sv_2, pst'', cst'', \varphi'', sky'', skies''}$$

$$\textbf{[ese-func-call]}\ \frac{\begin{array}{c} \overline{se}, pst, cst, \varphi, sky, skies \ \hat{\Downarrow}^{*}\ \overline{sv}, \langle lst', gst' \rangle, cst', \varphi', sky', skies' \\ \langle fst, gst' \rangle, cst', \varphi', \overline{s}, [S(m, d')], skies' \stackrel{\hat{}}{\Rightarrow} \langle fst', gst'' \rangle, cst'', \varphi'', [\,], fsky, skies'' \\ lookup(\#return, fst') = sv \end{array}}{\begin{array}{l} f(\overline{se})_l, pst, cst, \varphi, sky, skies \ \hat{\Downarrow}\ sv, \langle lst', gst'' \rangle, cst'', \varphi'', sky'', skies''' \\ \text{where} \\ f(\overline{x})\{_m\ \overline{s}\ \}_n \text{ is a defined function} \\ fst = [\overline{x} \mapsto \overline{sv}] \text{ is the initial local state for } f \\ d' = curDraw(c') \text{ is the power draw after evaluating the arguments} \\ d'' = curDraw(c'') \text{ is the power draw after executing } f \\ sky'' = sky' \mathbin{+\!\!+} [F(l), E(d'')] \\ skies''' = skies''[f \mapsto skies''(f) \cup \{fsky \mathbin{+\!\!+} [F(n)]\}] \end{array}}$$

$$\textbf{[ese-comp-call]}\ \frac{\delta_c(f, s_c) = \langle sv, \varphi', s_c' \rangle \qquad cst' = cst[c \mapsto s_c'] \qquad d = curDraw(cst')}{\begin{array}{l} c.f()_l, pst, cst, \varphi, sky, skies \ \hat{\Downarrow}\ sv, pst, cst', \varphi \wedge \varphi', sky', skies \\ \text{where } sky' = sky \mathbin{+\!\!+} [F(l), E(d)] \end{array}}$$

Figure 4.23: Energy-aware symbolic evaluation of expressions ($\hat{\Downarrow}$)

Part II

Formal Correctness

5 TopHat: A Formal Foundation for Task-Oriented Programming

Software that models how people work is omnipresent in today's society. Current languages and frameworks often focus on usability by non-programmers, sacrificing flexibility and high-level abstraction. Task-oriented programming (TOP) is a programming paradigm that aims to provide the desired level of abstraction while still being expressive enough to describe real-world collaboration. It prescribes a declarative programming style to specify multi-user workflows. Workflows can be higher-order. They communicate through typed values on a local and global level. Such specifications can be turned into interactive applications for different platforms, supporting collaboration during execution. TOP has been around for more than a decade, in the forms of iTasks and mTasks, which are tailored for real-world usability. So far, it has not been given a formalisation which is suitable for formal reasoning.

In this chapter we give a description of the TOP paradigm and then decompose its rich features into elementary language elements, which makes them suitable for formal treatment. We use the simply-typed lambda calculus, extended with pairs and references, as a base language. On top of this language, we develop TopHat, a language for modular interactive workflows. We describe TopHat by means of a layered semantics. These layers consist of multiple big-step evaluations on expressions, and two labelled transition systems, handling user inputs.

With TopHat we prepare a way to formally reason about TOP languages and programs. This approach allows for comparison with other work in the field. We have implemented the semantic rules of TopHat in Haskell, and the task layer on top of the iTasks framework. This shows that our approach is feasible, and lets us demonstrate the concepts by means of illustrative case studies. TOP has been applied in projects with the Dutch coast guard, tax office, and navy. Our work matters because formal program verification is important for mission-critical software, especially for systems with concurrency.

5.1 Introduction

Many applications these days are developed to support workflows in institutions and businesses. Take for example expense declarations, order processing, and emergency management. Some of these workflows occur on the boundary between organisations and customers, like flight bookings or tax returns. All these systems need to interact with different people (medical staff, tax officers, customers, etc.), and they use information from multiple sources (input forms, databases, sensors, etc.).

5.1.1 Tasks

Tasks are interactive units of work based on information sources. Tasks model real world collaboration between users, are driven by work users do, and are assigned to some user. Users could be people out in the field or sitting behind their desks, as well as machines doing calculations or fetching data.

5.1.2 Task-Oriented Programming

Task-oriented programming (TOP) is a programming paradigm that targets the sweet spot between faithful modelling workflows and rapid prototyping of multi-user web applications supporting these workflows [Plasmeijer et al., 2012]. TOP focusses on modelling collaboration patterns. This gives rise to a user's need to interact and share information. Next to that, TOP automatically provides solutions to common development jobs like designing GUIs, connecting to databases, and client-server communication.

Therefore, a language that supports TOP should choose the right level of abstraction to support two things. Firstly, it should provide primitive building blocks that are useful for high-level descriptions of how users collaborate, with each other and with machines. These building blocks are *editors*, *composition*, and *shared data*. Secondly, it should be able to generate applications, including graphical user interfaces, from workflows modelled with said building blocks.

Users can work together in a number of ways, and this is reflected in TOP by task compositions. There is *sequential* composition, *parallel* composition, and *choice*. Users need to communicate in order to engage in these forms of collaboration. This is reflected in TOP by three kinds of communication mechanisms. There is data flow *alongside* control flow, where the result of a task is passed onto the next. There is data flow *across* control flow, where information is shared between multiple tasks. Finally, there is communication with the *outside* world, where information is entered into the system via input events and output is returned via observations. The end points where the outside world interacts with TOP applications are called editors. In generated applications, editors can take many forms, like input fields, selection boxes, or map widgets.

5.1.3 Implementations of TOP

Currently, there are two frameworks that implement TOP: iTasks and mTasks. iTasks is an implementation of TOP, in the form of a shallowly embedded domain-specific language in the lazy functional programming language Clean. It is a library that provides editors, monadic combinators, and shared data sources. iTasks uses the generic programming facilities of Clean to derive rich client and server applications from a single source. It has been used to model an incident management tool for the Dutch coast guard [Lijnse et al., 2012]. Also, it has been used to prototype ideas for Command and Control systems [Kool, 2017; Stutterheim, 2017], and in a case study for the Dutch tax authority [Stutterheim et al., 2017]. mTasks is a subset of iTasks, focusing on IOT devices and deployment on microcontrollers. It has been used to control home thermostats and other home automation applications [Koopman et al., 2018].

5.1.4 Challenges

Both iTasks and mTasks have been designed for developing real-world applications. They are constantly being extended and improved with this goal in mind. The different variations of task combinators and the details that come with real-world requirements, make it hard to see what the essence of TOP is. In this chapter, we want to take a step back and look at the essence of TOP. We do this both formally and informally. Informally in the sense that we give an intuitive description of the features that define task-oriented programming. Formally in the sense that we develop a language which formalises these features as language constructs, and we give them a semantics in the style that is common in programming language research. We separate the task layer and the underlying host language, both syntactically and semantically. This makes it explicit which properties of TOP come from the task layer, and which come from functional programming. Our challenge, therefore, is to model the properties of TOP into a language and pave the way for formal treatment of TOP programs. We give this formal language the name $\widehat{\text{TOP}}$ (TopHat).

5.1.5 Contributions

Our contributions to workflow modelling, functional programming language design, and rapid application development are as follows.

1. We describe the essential concepts of task-oriented programming.
2. We present a language for modelling declarative workflows, called $\widehat{\text{TOP}}$, embedded in a simply-typed lambda calculus. It features the identified essential TOP concepts.
3. We develop a layered operational semantics for $\widehat{\text{TOP}}$ that is driven by user input. The semantics of the task language is clearly separated from the semantics of the underlying host language.
4. Along with this semantics, we present the following semantic observations on tasks: the current value, whether a term is stuck, the current user interface, and the accepted inputs.
5. We prove progress and type preservation for $\widehat{\text{TOP}}$.
6. Using both the essential concepts and the formal language, we compare TOP with related work in areas ranging from business process modelling, to process algebras and reactive programming.
7. We implemented the whole semantic system in Haskell [Marlow et al., 2010].
8. To create executable applications, we implemented the task layer of $\widehat{\text{TOP}}$ in iTasks. This also demonstrates that the former is a subset of the latter.

5.1.6 Structure

In section 5.2 we demonstrate the functionality of $\widehat{\text{TOP}}$ by means of an example, Section 5.3 gives an overview of the essential concepts of TOP. Section 5.4 introduces the $\widehat{\text{TOP}}$ language syntax and section 5.5 the semantics. Then in section 5.6 we show that certain properties hold for the language. We take a look at related work in section 5.7 and conclude in section 5.8.

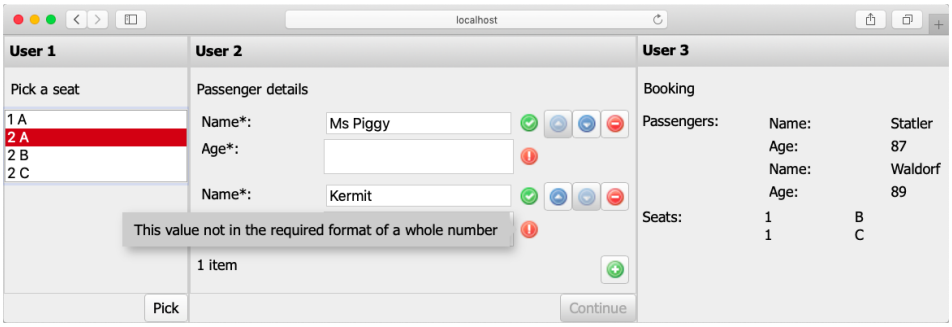


Figure 5.1: Running web application of the flight booking example using a translation to iTasks. It shows three users booking a flight simultaneously. The first user has entered name and age and continued picking seats. The second is entering details of two passengers. The ages are not filled in, therefore the Continue button is disabled. The message bubble shows that the age field only accepts integer values. The third user finished booking, which prevents the first user from picking seats 1B and 1C.

5.2 Example: A Flight Booking System

In this section we develop an example program to demonstrate the capabilities of $\widehat{\text{TOP}}$. The example is a small flight booking system. It demonstrates communication on all three levels: with the environment, across control flow, and alongside it. Also, it shows synchronisation and input validation. The requirements of the application are as follows.

1. A user has to enter a list of passengers for which to book tickets.
2. At least one of these passengers has to be an adult.
3. After a valid list of passengers has been entered, the user has to pick seats.
4. Only free seats may be picked.
5. Every passenger must have exactly one seat.
6. Multiple users should be able to book tickets at the same time.

For this example we assume that the host language has four functions over lists: all, any, intersect, and difference. The functions all and any check if all or any elements in a list satisfy a given predicate. The functions intersect and difference compute the set-intersection and set-difference of two lists. We also make use of string equality ($\equiv$), dereferencing (!), reference assignment ($:=$), and expression sequencing (;). For brevity, we omit the type annotations of variable bindings.

Example 5.2.1 (Flight booking)**.** We start off by defining some type aliases. A passenger is a pair of name and age. A seat is a pair of a row number and a seat letter.

```
type PASSENGER = STRING × INT
type SEAT = INT × STRING
```

Choosing seats requires reading and updating shared information. The list of free seats is stored in a reference.

```
let freeSeats = ref [⟨1,"A"⟩ , ⟨1,"B"⟩ , ⟨1,"C"⟩ , ...]
```

Now we develop our workflow in a top-down manner. Our flight booking starts with an interactive task ⊠(LIST PASSENGER), where users can enter a list of passengers. A task ⊠ τ is an empty editor that asks for a value of the given type τ. Passengers are valid if their name is not empty and their age is at least 0. Lists of passengers are valid if each passenger is valid, and at least one of the passengers is an adult. When the user has entered a valid list of passengers, the step after ▷ becomes enabled, and the user can proceed to picking seats. In case of an invalid list of passengers, the step is guarded by the failing task ↯ .

```
let valid = λp. not (fst p ≡ "") ∧ snd p ≥ 0 in
let adult = λp. snd p ≥ 18 in
let allValid = λps. all valid ps ∧ any adult ps in
let bookFlight = ⊠(LIST PASSENGER) ▷ λps.
  if allValid ps then chooseSeats ps else ↯
```

A selection of seats is correct if every entered seat is free.

```
let correct = λss. intersect ss !freeSeats ≡ ss in
let chooseSeats = λps. ⊠(LIST SEAT) ▷ λss.
  if correct ss ∧ length ps ≡ length ss
    then confirmBooking ps ss else ↯
```

The function confirmBooking removes the selected seats from the shared list of free seats, and displays the end result using an editor, denoted by □.

```
let confirmBooking = λps. λss.
  freeSeats := difference !freeSeats ss; □⟨ps, ss⟩
```

The main task starts three bookFlight tasks, which could be performed by three different users in parallel.

```
bookFlight ⋈ bookFlight ⋈ bookFlight
```

A screenshot of the running application is shown in fig. 5.1. All instances of the bookFlight task have access to the shared list of free seats. Rewriting the example in a language without side effects would not only be cumbersome, obfuscating the code with explicit threading of state, but it would be impossible to model the parallel execution of three bookFlight tasks. It is not known upfront which task will finish first, and thus it is not possible to thread the free seat list between the parallel tasks. □

5.3 Intuition

This section gives an overview of the abilities of tasks in the task-oriented programming paradigm.

5.3.1 Tasks Model Collaboration

The central objective of TOP is to *coordinate collaboration*. The basic building blocks of $\widehat{\text{TOP}}$ for expressing collaboration are task combinators. They express ways in which

people can work together. Tasks can be executed after each other, at the same time, or conditionally. This motivates the combinators step, parallel, and choice.

Example 5.3.1 (Breakfast)**.** The following program shows the different collaboration operators in the setting of making breakfast. Users have a choice (◊) whether they want tea or coffee. They always get an egg. The drink and the food are prepared in parallel (⋈). When both the drink and the food are prepared, users can step (▷) to eating the result.

```
let makeBreakfast : Task Drink → Task Food → Task ⟨Drink,Food⟩ =
  λmakeDrink. λmakeFood. makeDrink ⋈ makeFood in
makeBreakfast (makeTea ◊ makeCoffee) makeEgg ▷ enjoyBreakfast
```

The way the combinators are defined matches real life closely. When we want to have breakfast, we have to complete several other tasks first. We decide what we want to have and then prepare it. We can prepare the different items we have for breakfast in parallel, but not at the same time. For example, it is impossible to make scrambled eggs, and put on the kettle for tea simultaneously. Instead, what is meant by parallel is that *the order in which we do tasks and the smaller tasks that they are composed of, does not matter*. Then finally, only when every item we want to have for breakfast is ready, can we sit down and enjoy it. □

5.3.2 Tasks Are Reusable

There are three ways in which tasks are modular. First, larger tasks are composed of smaller ones. Second, tasks are first-class, they can be arguments and results of functions. Third, tasks can be result values of other tasks. These aspects make it possible for programmers to model custom collaboration patterns. Example 5.3.1 demonstrates how tasks can parameterised by other tasks: makeBreakfast is a collaboration pattern that always works the same way, regardless of which food and drink are being prepared.

5.3.3 Tasks Are Driven by User Input

Input events drive evaluation of tasks. The application of a valid event to the current task, results in a new task. This is how $\widehat{\text{TOP}}$ communicates with the environment. Inputs are synchronous, which means the order of execution is determined by the order of the inputs.

In $\widehat{\text{TOP}}$, *editors* are the basic method of communication with the environment. Editors are modelled after input widgets from graphical user interfaces. There are different editors, denoted by different box symbols. Take for example an editor holding the integer seven: □ 7. Such an editor reacts to change events, for example the values 42 or 37, which are of the same type. The sole purpose of editors is to interact with users by remembering the last value that has been sent to them. There are no output events. As values of editors can be observed, for example by a user interface, editors facilitate both input and output. An empty editor (⊠) stands for a prompt to input data, while a filled editor (□) can be seen either as outputting a value, or as an input that comes with a default value.

Example 5.3.2 (Vending machine)**.** The following example demonstrates the use of external communication and choice. We have a vending machine that dispenses a biscuit for one coin and a granola bar for two coins.

$$\boxtimes\textsc{Int} \rhd \lambda n.\ \textbf{if}\ n \equiv 1\ \textbf{then}\ \square\text{Biscuit}\ \textbf{else if}\ n \equiv 2\ \textbf{then}\ \square\text{GranolaBar}\ \textbf{else}\ \lightning$$

The editor $\boxtimes$ Int asks the user to enter an amount of money. This editor stands for a coin slot in a real machine that freely accepts and returns coins. There is a continue button that is initially disabled, due to the fact that the editor has no value. When the user has inserted exactly 1 or 2 coins, the continue button becomes enabled. When the user presses the continue button, the machine dispenses either a biscuit or a granola bar, depending on the amount of money. Snacks are modelled using a custom data type. □

5.3.4 Tasks Can Be Observed

Several observations can be made on tasks. One of those is determining the value of a task. Not all tasks have a value, which makes value observation partial. For example, the value of $\square\,7$ is 7, but the value of $\boxtimes$ Int is $\bot$. Another observation is the set of input events a task can respond to. For example, the task $\square\,7$ can respond to value events, as discussed before. To render a task, we need to observe a task's user interface. This is done compositionally. User interfaces of combined tasks are composed of the user interfaces of the components. For example, if two tasks combined with a step combinator, only the left-hand side is rendered. Two parallel tasks are rendered next to each other. Combining this information with the task's value and possible inputs, we can display the current state of the task, together with buttons that show the actions a user can engage in. The final observation is to determine whether a task fails, denoted by $\lightning$. The step combinator $\rhd$ and the choice combinator $\Diamond$ use this to prevent users from picking failing tasks.

5.3.5 Tasks Are Never Done

Tasks never terminate, they always keep reacting to events. Editors can always be changed, and step combinators move on to new tasks. In a step $t \rhd e$, the decision to move on from a task t to its continuation e is taken by $\rhd$, not by t. The decision is based on a speculative evaluation of e. The step combinator in $t \rhd e$ passes the value v of t to the continuation e. Steps act like t as long as the step is guarded. A step is guarded if either the left task has no value, or the speculative evaluation of e applied to v yields the failure task $\lightning$. Once it becomes unguarded, the step continues as the result of $e\ v$. Speculative evaluation is designed so that possible side effects are undone. Step combinators give rise to a form of internal communication. They represent data flow that *follows* control flow.

5.3.6 Tasks Can Share Information

The step combinator is one form of internal communication, where task values are passed to continuations. Another form of internal communication is shared data. Shared data enables data flow *across* control flow, in particular between parallel tasks. Shared data sources are assignable references whose changes are immediately visible to all tasks interested in them. Users cannot directly interact with shared data, a shared editor is required for that. If x is a reference of type τ, then $\blacksquare\,x$ is an editor whose value is that of x. The semantics of $\widehat{\text{TOP}}$ requires all updates to shared data and all enabled internal steps to be processed before any further communication with the environment can take place.

Example 5.3.3 (Cigarette smokers)**.** The cigarette smokers problem by Downey [2008] is a surprisingly tricky synchronisation problem. We study it here because it demonstrates the capabilities of guarded steps. The problem is stated as follows. To smoke a cigarette, three ingredients are required: tobacco, paper, and a match. There are three smokers, each having one of the ingredients and requiring the other two. There is an agent that randomly provides two of those. Downey models availability of the ingredients with a semaphore for each ingredient, and the agent randomly signals two of the three. The difficulty lies in the requirement that only the smoker may proceed whose missing ingredients are present. If one of the other smokers claims one of the ingredients, the system deadlocks. The solution proposed by Downey involves an additional mutex, three additional semaphores, three additional threads called *pushers*, and three regular Boolean variables. The job of the pushers is to record availability of their ingredient in their Boolean variable, and check availability of other resources, waking the correct smoker when appropriate. The solution to this problem, essentially deadlock-free waiting for two semaphores, requires a substantial amount of additional synchronisation, together with non-trivial conditional statements. $\widehat{\text{TOP}}$ allows a simple solution to this problem, using guarded steps. Steps can be guarded with arbitrary expressions. The parallel combinator can be used to watch two shared editors at the same time. Let match, paper, and tobacco be references to Booleans. The smokers are defined as follows.

```
let continue = λ⟨x,y⟩ . if x ∧ y then smoke else ↯ in
let tobaccoSmoker = (■ match ⋈ ■paper) ▷ continue in
let paperSmoker = (■ tobacco ⋈ ■match) ▷ continue in
let matchSmoker = (■ tobacco ⋈ ■paper) ▷ continue in
tobaccoSmoker ⋈ paperSmoker ⋈ matchSmoker
```

When the agent supplies two of the ingredients by setting the respective shares to True, only the step of the smoker that waits for those becomes enabled. □

5.3.7 Tasks Are Predictable

Let t_1 and t_2 be tasks. The parallel combination $t_1 \bowtie t_2$ stands for two independent tasks carried out at the same time. This operator introduces interleaving concurrency. For the system it does not matter if the tasks are executed by two different people, or by one person who switches between the tasks. The inputs sent to the component tasks are interleaved into a serial stream, which is sent to the parallel combinator. We assume that such a serialisation is always possible. The tasks are truly independent of each other, if all interleavings are possible. The environment prefixes events to t_1 and t_2 respectively by F (first) and S (second). This unambiguously renames the inputs, removing any source of nondeterminism.

With concurrency comes the need for synchronisation, in situations where only some but not all interleavings are desired. The basic method for synchronisation in $\widehat{\text{TOP}}$ is built into the step combinator. The task $t \rhd e$ can only continue execution when two conditions are met: Task t must have a value v, and $e\ v$ must not evaluate to $\lightning$. Programmers can encode arbitrary conditions in $e\ v$, which are evaluated atomically between interaction steps. This allows a variety of synchronisation problems to be solved in an intuitive and straight-forward manner. Hoare [1985] states that nondeterminism is only ever useful for *specifying* systems, never for implementing them. $\widehat{\text{TOP}}$ is meant solely for implementation and does not have

any form of nondeterminism. Input events for parallel tasks are disambiguated, internal steps (▶) have a well-defined evaluation order, and internal choice (⧫) is left-biased.

5.3.8 Recap

Collaboration in the real world consists of three aspects: communication, concurrency, and synchronisation. These aspects are reflected in TOP on a high level of abstraction, hiding the details of communication. For example, the cigarette smokers communicate with each other, but the programs do not explicitly mention sending or receiving events. By focusing on collaboration instead of communication, TOP leads to directly executable specifications closer to real-world workflows which, at the same time, can be used to generate multi-user applications to support such workflows. All abilities described in the previous section are captured by the three building blocks of tasks: *editors*, *composition*, and *shared data*. Editors facilitate interaction and are the observable part of tasks. Combinators are at the heart of modelling collaboration. They describe what needs to be done, in which order. Finally, shared data facilitates communication between tasks. These three building blocks will be formalised in the next section.

5.4 Language

In this section, we present the constructs of $\widehat{\text{TOP}}$, our modular interactive workflow language. We define the host and task language, the types, and the static semantics. Then we describe the workings of each construct using examples. These constructs are formalised in section 5.5.

5.4.1 Expressions

The host language is a simply-typed lambda calculus, extended with some basic types and ML-style references. We use references to represent shared data sources. The simply-typed lambda calculus does not support recursion. The grammar in fig. 5.2 defines the syntax of the host language. It has abstractions, applications, variables, and constants for booleans, integers and strings. The symbol $\star$ stands for binary operators. For the result of parallel tasks we need pairs. Conditionals come in handy for defining guards. References will be used to implement shared editors. Our treatment of references closely follows the one by Pierce [2002]. Creating a reference using the keyword **ref** yields a location l. x denotes program variables, l denotes store locations. Locations are not intended to be directly manipulated by the programmer. The symbols ! and := stand for dereferencing and assignment. The unit value is used as the result of assignments.

Notation. We use double quotation marks to denote strings. Integers are denoted by their numerical representation, and booleans are written True and False. We freely make use of the logic operators $\neg$, $\wedge$, and $\vee$, arithmetic operators $+$, $-$, $\times$, $/$, and the string append operator $+\!\!+$. Furthermore, we use standard comparison operations $<$, $\leq$, $\equiv$, $\not\equiv$, $\geq$, and $>$. The symbol $\star$ stands for any of those. The notation $e_1; e_2$ is an abbreviation for $(\lambda x : \textsc{Unit}.\ e_2)\ e_1$, where x is a fresh variable. The notation $\textbf{let}\, x : \tau = e_1\, \textbf{in}\, e_2$ is an abbreviation for $(\lambda x : \tau.\ e_2)\ e_1$.

Expressions	e	::=	$\lambda x : \tau.\, e \mid e_1\ e_2$	– abstraction, application
		|	$x \mid c \mid e_1 \star e_2$	– variable, constant, operation
		|	$\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \langle\rangle$	– branch, unit
		|	$\langle e_1, e_2\rangle \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e$	– pair, projections
		|	$[\,]_\beta \mid e_1 :: e_2$	– nil, cons
		|	$\mathsf{head}\ e \mid \mathsf{tail}\ e$	– first element, list tail
		|	$\textbf{ref}\, e \mid !e \mid e_1 := e_2 \mid l$	– references, location
		|	p	– pretask
Constants	c	::=	$B \mid I \mid S$	– boolean, integer, string

Figure 5.2: Language grammar

Pretasks	p	::=	$\square\, e \mid \boxtimes\, \beta \mid \blacksquare\, e$	– editors: valued, unvalued, shared
		|	$e_1 \blacktriangleright e_2 \mid e_1 \triangleright e_2$	– steps: internal, external
		|	$\lightning \mid e_1 \bowtie e_2$	– fail, combination
		|	$e_1 \blacklozenge e_2 \mid e_1 \lozenge e_2$	– choice: internal, external

Figure 5.3: Task grammar

Types	τ	::=	$\tau_1 \to \tau_2 \mid \beta \mid \textsc{Ref}\ \tau \mid \textsc{Task}\ \tau$	– function, basic, reference, task
Basic types	β	::=	$\tau_1 \times \tau_2 \mid \textsc{List}\ \beta \mid \textsc{Unit}$	– product, list, unit
		|	$\textsc{Bool} \mid \textsc{Int} \mid \textsc{String}$	– Boolean, integer, string

Figure 5.4: Type grammar

Pretasks. The grammar in fig. 5.3 specifies the syntactic category of *pretasks*. Pretasks are tasks that contain unevaluated subexpressions. Each pretask will be discussed in more detail in the following subsections. We use open symbols ($\square$, $\boxtimes$, $\triangleright$, $\lozenge$) for tasks that require user input, and closed symbols ($\blacksquare$, $\blacktriangleright$, $\blacklozenge$) for tasks that can be evaluated without user input.

Typing. Figure 5.4 shows the grammar of types used by $\widehat{\textsc{top}}$. It has functions, pairs, lists, basic types, unit, references, and tasks. The typing rules for expressions are given in fig. 5.5. Typing rules are of the form $\Gamma, \Sigma \vdash e : \tau$, which should be read as "in environment Γ and store typing Σ, expression e has type τ". Most typing rules lift the type of their subexpressions into the Task-type. The typing rules for steps (**[T-Then]**, **[T-Next]**) make sure the continuations e_2 are functions that accept a well-typed value from the left-hand side. References, and therefore shared editors (**[T-Update]**), can only be of basic type so they do not introduce implicit recursion.

5.4.2 Editors

Programs in $\widehat{\textsc{top}}$ model interactive workflows. Interaction means that end users should be able to enter information into the system, change it, and so on. To do this, we introduce the concept of *editors*. Editors are typed containers that either hold a value or are empty. Editors that have a value can be *changed*. Empty editors can be *filled*. Editors are used for

$$\boxed{\Gamma, \Sigma \vdash e : \tau}$$

[T-ConstBool]
$$\frac{c \in B}{\Gamma, \Sigma \vdash c : \textsc{Bool}}$$

[T-ConstInt]
$$\frac{c \in I}{\Gamma, \Sigma \vdash c : \textsc{Int}}$$

[T-ConstString]
$$\frac{c \in S}{\Gamma, \Sigma \vdash c : \textsc{String}}$$

[T-Var]
$$\frac{x : \tau \in \Gamma}{\Gamma, \Sigma \vdash x : \tau}$$

[T-Unit]
$$\frac{}{\Gamma, \Sigma \vdash \langle\rangle : \textsc{Unit}}$$

[T-Loc]
$$\frac{\Sigma(l) = \beta}{\Gamma, \Sigma \vdash l : \textsc{Ref}\ \beta}$$

[T-Pair]
$$\frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_2}{\Gamma, \Sigma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

[T-First]
$$\frac{\Gamma, \Sigma \vdash e : \tau_1 \times \tau_2}{\Gamma, \Sigma \vdash \mathsf{fst}\, e : \tau_1}$$

[T-Second]
$$\frac{\Gamma, \Sigma \vdash e : \tau_1 \times \tau_2}{\Gamma, \Sigma \vdash \mathsf{snd}\, e : \tau_2}$$

[T-ListEmpty]
$$\frac{}{\Gamma, \Sigma \vdash [\,]_\beta : \textsc{List}\ \beta}$$

[T-ListCons]
$$\frac{\Gamma, \Sigma \vdash e_1 : \beta \quad \Gamma, \Sigma \vdash e_2 : \textsc{List}\ \beta}{\Gamma, \Sigma \vdash e_1 :: e_2 : \textsc{List}\ \beta}$$

[T-ListHead]
$$\frac{\Gamma, \Sigma \vdash e : \textsc{List}\ \beta}{\Gamma, \Sigma \vdash \mathsf{head}\, e : \beta}$$

[T-ListTail]
$$\frac{\Gamma, \Sigma \vdash e : \textsc{List}\ \beta}{\Gamma, \Sigma \vdash \mathsf{tail}\, e : \textsc{List}\ \beta}$$

[T-Abs]
$$\frac{\Gamma[x : \tau_1], \Sigma \vdash e : \tau_2}{\Gamma, \Sigma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}$$

[T-If]
$$\frac{\Gamma, \Sigma \vdash e_1 : \textsc{Bool} \quad \Gamma, \Sigma \vdash e_2 : \tau \quad \Gamma, \Sigma \vdash e_3 : \tau}{\Gamma, \Sigma \vdash \textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3 : \tau}$$

[T-App]
$$\frac{\Gamma, \Sigma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma, \Sigma \vdash e_2 : \tau_1}{\Gamma, \Sigma \vdash e_1 e_2 : \tau_2}$$

[T-Ref]
$$\frac{\Gamma, \Sigma \vdash e : \beta}{\Gamma, \Sigma \vdash \textbf{ref}\, e : \textsc{Ref}\ \beta}$$

[T-Deref]
$$\frac{\Gamma, \Sigma \vdash e : \textsc{Ref}\ \beta}{\Gamma, \Sigma \vdash !e : \beta}$$

[T-Assign]
$$\frac{\Gamma, \Sigma \vdash e_1 : \textsc{Ref}\ \beta \quad \Gamma, \Sigma \vdash e_2 : \beta}{\Gamma, \Sigma \vdash e_1 := e_2 : \textsc{Unit}}$$

[T-Fail]
$$\frac{}{\Gamma, \Sigma \vdash \lightning : \textsc{Task}\ \tau}$$

[T-Edit]
$$\frac{\Gamma, \Sigma \vdash e : \beta}{\Gamma, \Sigma \vdash \square e : \textsc{Task}\ \beta}$$

[T-Enter]
$$\frac{}{\Gamma, \Sigma \vdash \boxtimes\, \beta : \textsc{Task}\ \beta}$$

[T-Update]
$$\frac{\Gamma, \Sigma \vdash e : \textsc{Ref}\ \beta}{\Gamma, \Sigma \vdash \blacksquare\, e : \textsc{Task}\ \beta}$$

[T-Or]
$$\frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\ \tau \quad \Gamma, \Sigma \vdash e_2 : \textsc{Task}\ \tau}{\Gamma, \Sigma \vdash e_1 \blacklozenge e_2 : \textsc{Task}\ \tau}$$

[T-Then]
$$\frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\ \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_1 \to \textsc{Task}\ \tau_2}{\Gamma, \Sigma \vdash e_1 \blacktriangleright e_2 : \textsc{Task}\ \tau_2}$$

[T-Next]
$$\frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\ \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \tau_1 \to \textsc{Task}\ \tau_2}{\Gamma, \Sigma \vdash e_1 \triangleright e_2 : \textsc{Task}\ \tau_2}$$

[T-And]
$$\frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\ \tau_1 \quad \Gamma, \Sigma \vdash e_2 : \textsc{Task}\ \tau_2}{\Gamma, \Sigma \vdash e_1 \bowtie e_2 : \textsc{Task}\ (\tau_1 \times \tau_2)}$$

[T-Xor]
$$\frac{\Gamma, \Sigma \vdash e_1 : \textsc{Task}\ \tau \quad \Gamma, \Sigma \vdash e_2 : \textsc{Task}\ \tau}{\Gamma, \Sigma \vdash e_1 \Diamond e_2 : \textsc{Task}\ \tau}$$

Figure 5.5: Typing rules

various forms of input and output, for example widgets in a GUI, form fields on a webpage, sensors, or network connections. Consider an editor for a person's age on a web page. Users can change the value until they are satisfied with it. Editors are meant to capture this constantly changing nature of user input. The user interface of an editor depends on its type. This could be an input field for strings, a toggle switch for booleans, or even a map with a pin for locations.

Valued and unvalued editors ($\square\, e, \boxtimes\, \beta$). Editors that hold an expression $e : \beta$ have type $\textsc{Task}\ \beta$. Empty editors are annotated with a type to ensure type safety and type preservation during evaluation.

Shared editors ($\blacksquare\, e$). Shared editors watch references, lifting their value into the task domain. If e is a reference $\textsc{Ref}\ \beta$, then $\blacksquare\, e$ is of type $\textsc{Task}\ \beta$. Changes to a shared editor are immediately visible to all shared editors watching the same reference. Imagine two users, Marco and Christopher, both watching shared editors of the same GPS coordinates. The editors are visualised as a pin on a map. When Marco moves his pin, he updates the value of the shared editor, thereby changing the value of the reference. This change is immediately reflected on Christopher's screen: The pin changes its position on his map. This way Marco and Christopher can work together to edit the same information.

Two other important use cases for shared editors are sensors and time. Sensors can be represented as external entities that periodically update a shared editor with their current sensor value. Similarly, the current time can be stored in a shared editor $\blacksquare\mathsf{time}$ that is periodically updated by a clock. The actual sensor and the clock are not modelled in $\widehat{\text{TOP}}$. We assume that they exist as external users that send update events to the system. This allows programmers to write tasks that react to sensor values or timeouts.

5.4.3 Steps

Editors represent atomic units of work. In this section we look at ways to compose smaller tasks into bigger ones. Composing tasks can be done in two ways, sequential and parallel. Parallel composition comes in two variants: combining two tasks (*and*-parallel) and choosing between two tasks (*or*-parallel). We study sequential composition first, and after that combining and choosing.

Internal and external step ($t \blacktriangleright e, t \triangleright e$). Sequential composition has a task t on the left and a continuation e on the right. External steps ($\triangleright$) are triggered by the user, while internal steps ($\blacktriangleright$) are taken automatically. Their typing rules are **[T-Then]** and **[T-Next]**. According to these rules, the left-hand side is a task $t : \textsc{Task}\ \tau_1$, and the right-hand side $e : \tau_1 \to \textsc{Task}\ \tau_2$ is a function that, given the task value of t, calculates the task with which to continue.

Steps are guarded, which means that the step combinators can only proceed when the following conditions are met. The left-hand side must have a value, only then can the right-hand side calculate the successor task. The successor task must not be $\lightning$, introduced below. This is enforced on the semantic level, as described in the next section. The internal step can proceed immediately when these conditions are met. The external step must additionally receive a continue event C.

Example 5.4.1 (Conditional stepping)**.** Consider the following program.

$$\boxtimes\textsc{Int} \blacktriangleright \lambda n.\ \textbf{if}\ n \equiv 42\ \textbf{then}\ \square\text{"Good"}\ \textbf{else}\ \square\text{"Bad"}$$

Initially, the step is guarded because the editor does not have a value. When a user enters an integer, the program continues immediately with either $\square$"Good" or $\square$"Bad", depending on the input.

Fail ($\lightning$). Fail is a task that never has a value and never accepts input. The typing rule **[T-Fail]** states that it has type $\textsc{Task}\ \tau$ for any type τ. Programmers can use $\lightning$ to tell steps that no sensible successor task can proceed.

Example 5.4.2 (Guarded stepping)**.** Consider this slight variation of example 5.4.1.

$$\boxtimes\textsc{Int} \blacktriangleright \lambda n.\ \textbf{if}\ n \equiv 42\ \textbf{then}\ \square\text{"Good"}\ \textbf{else}\ \lightning$$

The user is asked to enter an integer. As long as the right-hand side of $\blacktriangleright$ evaluates to $\lightning$, the step cannot proceed, and the user can keep editing the integer. As soon as the value of the left-hand side is 42, the right-hand side evaluates to something other than $\lightning$, and the step proceeds to $\square$"Good". □

Example 5.4.3 (Waiting)**.** With the language constructs seen so far it is possible to create a task that waits for a specified amount of time. To do this, we make use of a shared editor holding the current time, and a guarded internal step.

$$\begin{aligned}&\textbf{let}\ \text{wait} : \textsc{Int} \to \textsc{Task}\ \textsc{Unit} = \lambda \mathit{amount} : \textsc{Int}.\\ &\quad \blacksquare\text{time} \blacktriangleright \lambda \mathit{start} : \textsc{Int}.\\ &\quad \blacksquare\text{time} \blacktriangleright \lambda \mathit{now} : \textsc{Int}.\\ &\qquad \textbf{if}\ \mathit{now} > \mathit{start} + \mathit{amount}\ \textbf{then}\ \square\langle\rangle\ \textbf{else}\ \lightning\end{aligned}$$

The first step is immediately taken, resulting in *start* to be the time at the moment wait is executed. The second step is guarded until the current time is greater to the start time plus the required *amount*. □

5.4.4 Parallel

A common pattern in workflow design is splitting up work into multiple tasks that can be executed simultaneously. In $\widehat{\text{TOP}}$, all parallel branches can progress independently, driven by input events. This requires inputs to be tagged in order to reach the intended task. There are two ways to proceed after a parallel composition. One way is to wait for all tasks to produce results and combine those, the other to pick the first available result. Both ways introduce explicit forks and implicit joins in $\widehat{\text{TOP}}$.

Combination ($e_1 \bowtie e_2$). A combination of two tasks is a parallel *and*. It has a value only if both branches have a value. This is reflected in the typing rule **[T-And]**, which shows that if the first task has type τ_1, and the second has type τ_2, their combination has type $\tau_1 \times \tau_2$.

Example 5.4.4 (Combining)**.** The task

$$\boxtimes\textsc{Int} \bowtie \square\text{" Batman"} \blacktriangleright \lambda\langle n, s\rangle\ .\ \square(\text{replicate}\ n\ \text{"Na"} +\!\!+\ s)$$

can only step when both editors have values. When it steps, the continuation uses the pair to calculate the result. □

Internal and external choice ($e_1 \blacklozenge e_2, e_1 \diamondsuit e_2$). Internal choice ($\blacklozenge$) is a parallel *or*. It picks the leftmost branch that has a value. Its typing rule **[T-Or]** states that both branches must have the same type $\textsc{Task}\ \tau$. For example, $\boxtimes\ \textsc{Int} \blacklozenge \square\ 37$ normalises to $\square\ 37$, because $\boxtimes\ \textsc{Int}$ doesn't have a value. Users can work on both branches of an internal choice simultaneously. External choice ($\diamondsuit$) is different in that it requires users to pick a branch before continuing with it. This means users cannot work on the branches of $\diamondsuit$ before picking one.

Example 5.4.5 (Delay)**.** We illustrate the use of internal and external choice by means of an example that asks a user to proceed with a given task or to cancel. If the user does not make a choice within a given time frame, the program proceeds automatically. The example makes use of the task wait from example 5.4.3.

```
let cancel : Task Unit = □⟨⟩ in
let delay : Int → Task Unit → Task Unit = λn. λproceed.
  (proceed ◇ cancel) ♦ (wait n ▶ λu : Unit. proceed)
```

□

5.4.5 Annotations

Tasks can be annotated with additional information. The system can use this information in various ways. Possible use cases are labels for the user interface, resource consumption information for static resource analysis, or messages for automatic end user feedback. Annotations are not covered in this chapter. Our Haskell implementation of $\widehat{\text{TOP}}$ supports annotating tasks with user IDs, so that individual tasks in a large workflow can be assigned to different users. These annotations are used to filter the user interfaces for each user so that they can only see their part of the workflow.

5.5 Semantics

In this section we formalise the semantics of the language constructs described in section 5.4. We organise this by following the structure of the language. Firstly, the task language is embedded in a simply-typed lambda calculus. This requires a specification of the *evaluation* of terms in the host language, and how it handles the task language. Secondly, there are two ways to drive evaluation of task expressions, internally by the system itself, and externally by the user. This is done in two additional semantics, one for the internal *normalisation* of tasks, and another for the *interaction* with the end user. The roles of the semantics can be summarized as follows.

evaluate	Reduce an expression until its normal form is a task.
normalize	Given a task, reduce all internal steps and choices until a task is reached that requires user input.
interact	Given a task that requires user input, process a single input event and reduce until the program again requires user input.

The three main layers of semantics are thus evaluation, normalisation, and interaction. The semantics, together with *observations*, will be discussed in the following subsections. Figure 5.6 shows the relation between all semantics arrows. It also shows that there are two

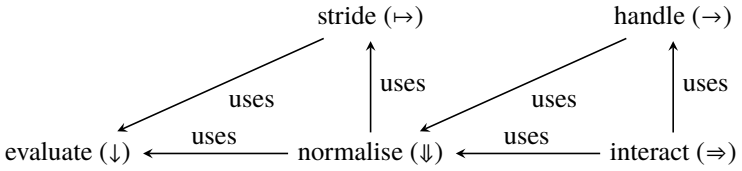


Figure 5.6: Semantic functions defined in this report and their relation

Values	v	::=	$\lambda x : \tau.\ e \mid \langle v_1, v_2 \rangle \mid \langle\rangle$	– abstraction, pair, unit
		|	$c \mid l \mid t$	– constant, location, task
Tasks	t	::=	$\square\, v \mid \boxtimes\, \tau \mid \blacksquare\, l$	– editors
		|	$t_1 \blacktriangleright e_2 \mid t_1 \triangleright e_2$	– steps
		|	$\lightning \mid t_1 \bowtie t_2$	– fail, combination
		|	$t_1 \blacklozenge t_2 \mid e_1 \lozenge e_2$	– choices

Figure 5.7: Value grammar

helper semantics, *handle* and *stride*. We use the convention that downward arrows are big-step semantics, and rightward arrows are small-step semantics. One of our explicit goals is to keep the semantics for evaluation and normalisation separate, to not mix general-purpose programming notions with workflow-specific semantics. This is achieved by letting tasks be values in the host language.

5.5.1 Evaluating Expressions

The host language evaluates expressions using a big-step semantics. To ease reasoning about references, we choose a call-by-value evaluation strategy. Figure 5.7 shows values that are the result of the evaluation semantics. Tasks are values, and the operands of task constructors are evaluated eagerly. Exceptions to this are steps and external choice, where some or all of the operands are not evaluated.

The rules to evaluate expressions are listed in fig. 5.8. Most rules do not differ from standard rules for evaluating expressions in the simply-typed lambda calculus [Pierce, 2002], except for the task constructs. Most task constructors are strict in their arguments. Only steps keep their right-hand side unevaluated to delay side effects until the step is taken. The same holds for both branches of external choices.

5.5.2 Task Observations

The normalisation (⇓) and interaction (⇒) semantics make use of observations on tasks. Observations are semantic functions on the syntax tree of tasks. There are four semantic functions: $\mathscr{V}$ for the current task value, $\mathscr{F}$ to determine if a task fails, $\mathscr{T}$ for the currently accepted input events, and a function for generating user interfaces. The semantics make use of $\mathscr{V}$ and $\mathscr{F}$, while $\mathscr{T}$ is used for proving safety. The function for user interfaces is not used by the semantics, but by our implementation. It is only described in passing here. The function $\mathscr{T}$ is described in section 5.5.4.

$$\boxed{e, \sigma \downarrow v, \sigma'}$$

[E-Value]

$$\frac{}{v, \sigma \downarrow v, \sigma}$$

[E-Pair]

$$\frac{e_1, \sigma \downarrow v_1, \sigma' \qquad e_2, \sigma' \downarrow v_2, \sigma''}{\langle e_1, e_2 \rangle, \sigma \downarrow \langle v_1, v_2 \rangle, \sigma''}$$

[E-First]

$$\frac{e, \sigma \downarrow \langle v_1, v_2 \rangle, \sigma'}{\mathsf{fst}\, e, \sigma \downarrow v_1, \sigma'}$$

[E-Second]

$$\frac{e, \sigma \downarrow \langle v_1, v_2 \rangle, \sigma'}{\mathsf{snd}\, e, \sigma \downarrow v_2, \sigma'}$$

[E-Cons]

$$\frac{e_1, \sigma \downarrow v_1, \sigma' \qquad e_2, \sigma' \downarrow v_2, \sigma''}{e_1 :: e_2, \sigma \downarrow v_1 :: v_2, \sigma''}$$

[E-Head]

$$\frac{e, \sigma \downarrow v_1 :: v_2, \sigma'}{\mathsf{head}\, e, \sigma \downarrow v_1, \sigma'}$$

[E-Tail]

$$\frac{e, \sigma \downarrow v_1 :: v_2, \sigma'}{\mathsf{tail}\, e, \sigma \downarrow v_2, \sigma'}$$

[E-App]

$$\frac{e_1, \sigma \downarrow \lambda x : \tau . e_1', \sigma' \qquad e_2, \sigma' \downarrow v_2, \sigma'' \qquad e_1'[x \mapsto v_2], \sigma'' \downarrow v_1, \sigma'''}{e_1 e_2, \sigma \downarrow v_1, \sigma'''}$$

[E-IfTrue]

$$\frac{e_1, \sigma \downarrow \mathsf{True}, \sigma' \qquad e_2, \sigma' \downarrow v_2, \sigma''}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3, \sigma \downarrow v_2, \sigma''}$$

[E-IfFalse]

$$\frac{e_1, \sigma \downarrow \mathsf{False}, \sigma' \qquad e_3, \sigma' \downarrow v_3, \sigma''}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3, \sigma \downarrow v_3, \sigma''}$$

[E-Ref]

$$\frac{e, \sigma \downarrow v, \sigma' \qquad l \notin Dom(\sigma')}{\textbf{ref}\, e, \sigma \downarrow l, \sigma'[l \mapsto v]}$$

[E-Deref]

$$\frac{e, \sigma \downarrow l, \sigma'}{!e, \sigma \downarrow \sigma'(l), \sigma'}$$

[E-Assign]

$$\frac{e_1, \sigma \downarrow l, \sigma' \qquad e_2, \sigma' \downarrow v_2, \sigma''}{e_1 := e_2, \sigma \downarrow \langle\rangle, \sigma''[l \mapsto v_2]}$$

[E-Edit]

$$\frac{e, \sigma \downarrow v, \sigma'}{\square\, e, \sigma \downarrow \square\, v, \sigma'}$$

[E-Update]

$$\frac{e, \sigma \downarrow l, \sigma'}{\blacksquare\, e, \sigma \downarrow \blacksquare\, l, \sigma'}$$

[E-Then]

$$\frac{e_1, \sigma \downarrow t_1, \sigma'}{e_1 \blacktriangleright e_2, \sigma \downarrow t_1 \blacktriangleright e_2, \sigma'}$$

[E-Next]

$$\frac{e_1, \sigma \downarrow t_1, \sigma'}{e_1 \triangleright e_2, \sigma \downarrow t_1 \triangleright e_2, \sigma'}$$

[E-And]

$$\frac{e_1, \sigma \downarrow t_1, \sigma' \qquad e_2, \sigma' \downarrow t_2, \sigma''}{e_1 \bowtie e_2, \sigma \downarrow t_1 \bowtie t_2, \sigma''}$$

[E-Or]

$$\frac{e_1, \sigma \downarrow t_1, \sigma' \qquad e_2, \sigma' \downarrow t_2, \sigma''}{e_1 \blacklozenge e_2, \sigma \downarrow t_1 \blacklozenge t_2, \sigma''}$$

Figure 5.8: Evaluation semantics for expressions

$$
\begin{array}{lcl}
\mathscr{V} : \text{Task} \times \text{State} \rightharpoonup \text{Value} & & \\
\mathscr{V}(\square\, v, \sigma) & = & v \\
\mathscr{V}(\boxtimes\, \beta, \sigma) & = & \bot \\
\mathscr{V}(\blacksquare\, l, \sigma) & = & \sigma(l) \\
\mathscr{V}(\lightning, \sigma) & = & \bot \\
\mathscr{V}(t_1 \blacktriangleright e_2, \sigma) & = & \bot \\
\mathscr{V}(t_1 \triangleright e_2, \sigma) & = & \bot \\
\mathscr{V}(t_1 \bowtie t_2, \sigma) & = & \begin{cases} \langle v_1, v_2 \rangle & \textbf{when } \mathscr{V}(t_1, \sigma) = v_1 \wedge \mathscr{V}(t_2, \sigma) = v_2 \\ \bot & \textbf{otherwise} \end{cases} \\
\mathscr{V}(t_1 \blacklozenge t_2, \sigma) & = & \begin{cases} v_1 & \textbf{when } \mathscr{V}(t_1, \sigma) = v_1 \\ v_2 & \textbf{when } \mathscr{V}(t_1, \sigma) = \bot \wedge \mathscr{V}(t_2, \sigma) = v_2 \\ \bot & \textbf{otherwise} \end{cases} \\
\mathscr{V}(t_1 \lozenge t_2, \sigma) & = & \bot
\end{array}
$$

Figure 5.9: Values observation function

$$
\begin{array}{lcl}
\mathscr{F} : \text{Task} \times \text{State} \to \text{Bool} & & \\
\mathscr{F}(\square\, v, \sigma) & = & \mathsf{False} \\
\mathscr{F}(\boxtimes\, \beta, \sigma) & = & \mathsf{False} \\
\mathscr{F}(\blacksquare\, l, \sigma) & = & \mathsf{False} \\
\mathscr{F}(\lightning, \sigma) & = & \mathsf{True} \\
\mathscr{F}(t_1 \blacktriangleright e_2, \sigma) & = & \mathscr{F}(t_1, \sigma) \\
\mathscr{F}(t_1 \triangleright e_2, \sigma) & = & \mathscr{F}(t_1, \sigma) \\
\mathscr{F}(t_1 \bowtie t_2, \sigma) & = & \mathscr{F}(t_1, \sigma) \wedge \mathscr{F}(t_2, \sigma) \\
\mathscr{F}(t_1 \blacklozenge t_2, \sigma) & = & \mathscr{F}(t_1, \sigma) \wedge \mathscr{F}(t_2, \sigma) \\
\mathscr{F}(e_1 \lozenge e_2, \sigma) & = & \mathscr{F}(t_1, \sigma_1') \wedge \mathscr{F}(t_2, \sigma_2') \\
 & & \quad \textbf{where } e_1, \sigma \Downarrow t_1, \sigma_1' \textbf{ and } e_2, \sigma \Downarrow t_2, \sigma_2'
\end{array}
$$

Figure 5.10: Failing observation function

Observable values ($\mathscr{V}$). Steps use task values to calculate the successor task. The function $\mathscr{V}$ associates a value v to task t where possible. Its definition is given in fig. 5.9. We use a half arrow ($\rightharpoonup$) to indicate that this function is partial, and $\bot$ to indicate when the function is undefined. Filled editors are tasks that contain values, as are shared editors. Unvalued editors do not contain values, neither does the fail task. These facts propagate through all other task constructors. Internal and external steps do not have an observable value, because calculating the value would require evaluation of the continuation. Parallel composition only has a value when both branches have values, in which case these values are paired. Internal choice has a value when one of the branches has a value. When both branches have a value, it takes the value of the left branch. External choice does not have a value because it waits for user input.

Failing ($\mathscr{F}$). The task *fail* ($\lightning$) stands for a failing task. A task is failing if there is no interaction possible, and there is no observable task value. Combinations of tasks can also be failing, for example the parallel composition of two fails ($\lightning \bowtie \lightning$). This expression is equivalent to $\lightning$, because it cannot handle input and cannot be further normalised.

$$\boxed{e, \sigma \Downarrow t, \sigma'}$$

[N-Done]

$$\frac{e, \sigma \downarrow t, \sigma' \qquad t, \sigma' \mapsto t', \sigma'' \qquad \sigma' = \sigma'' \wedge t = t'}{e, \sigma \Downarrow t, \sigma'}$$

[N-Repeat]

$$\frac{e, \sigma \downarrow t, \sigma' \qquad t, \sigma' \mapsto t', \sigma'' \qquad \sigma' \neq \sigma'' \vee t \neq t' \qquad t', \sigma'' \Downarrow t'', \sigma'''}{e, \sigma \Downarrow t'', \sigma'''}$$

Figure 5.11: Normalisation semantics

The function $\mathscr{F}$ in fig. 5.10 tells whether a given task is failing. Steps whose left-hand sides are failing can never proceed because of the lack of an observable value. Therefore, they are themselves failing. The parallel combination of two tasks is failing if they are both failing. If only one of them fails, there is still interaction possible with the other task. Internal choices with two failing tasks are failing. External choices let the user pick a side and only then evaluate the corresponding subexpression. To determine if an external choice is failing, it needs to peek into the future to check if both subexpressions are failing. The choice case relies on the normalisation semantics ($\Downarrow$) defined in the next section.

User interface. $\widehat{\text{TOP}}$ is designed such that a user interface can be generated from a task's syntax tree. A possible graphical user interface is shown in fig. 5.1, where tasks are rendered as HTML pages. Editors are rendered as input fields, external choices are represented by two buttons, and parallel tasks are rendered side by side. Steps only show the interface of their left-hand side. In case of an external step they are accompanied by a button. When the guard condition of a step is not fulfilled, the button is disabled.

5.5.3 Normalising Tasks

The normalisation semantics is responsible for reducing expressions of type Task until they are ready to handle input. It is a big-step semantics, and makes use of evaluation of the host language. We write $e, \sigma \Downarrow t, \sigma'$ to describe that an expression e in state σ normalises to task t in state σ'. Normalisation rules are given in fig. 5.11. Both rules ensure that expressions are first evaluated by the host language ($\downarrow$), and then by the stride semantics ($\mapsto$). These two actions are repeated until the resulting state and task stabilise.

The striding semantics is responsible for reducing internal steps and internal choices. A stride from task t in state σ to t' in state σ' is denoted by $t, \sigma \mapsto t', \sigma'$. The rules for striding are given in fig. 5.12. Tasks like editors, fail and external choice are not further reduced. For external choice and parallel there are congruence rules.

The split between striding and normalisation is due to mutable references. Consider the following example, where $\sigma = \{ s \mapsto \mathsf{False} \}$.

$$(\blacksquare\ s \blacktriangleright \lambda x{:}\text{Bool}.\ \textbf{if}\ x\ \textbf{then}\ e\ \textbf{else}\ \lightning\ ) \bowtie (s := \mathsf{True};\ \square\langle\rangle\ )$$

[S-And] reduces this expression in one step to:

$$(\blacksquare\ s \blacktriangleright \lambda x{:}\textsc{Bool}.\ \textbf{if}\ x\ \textbf{then}\ e\ \textbf{else}\ \lightning\ ) \bowtie (\square\ \langle\rangle\ )$$

with $\sigma' = \{\, s \mapsto \mathsf{True}\,\}$. This expression is not normalised, because the left task can take a step. The issue here lies in the fact that the right task updates l. The **[N-Done]** and **[N-Repeat]** rules ensure that striding is applied until the state σ becomes stable and no further normalisation can take place.

Principles of stepping. Considering the expression $t_1 \blacktriangleright e$, stepping away from task t_1 can only be performed when t_1 has a value: $\mathscr{V}(t_1) = v_1$. Only then can a new task t_2 be calculated from the application of the result value v_1 to the expression e. On top of that, t_2 must not be failing: $\neg\mathscr{F}(t_2)$. These principles lead to the stepping rules in fig. 5.12. **[S-ThenStay]** does nothing, because the left side does not have a value. **[S-ThenFail]** covers the case that the left side has a value but the calculated successor task is failing. This rule uses the semantics of the host language to evaluate the application $e_2 v_1$. When all required conditions are fulfilled, **[S-ThenCont]** allows stepping to the successor task.

Principles of choosing. Choosing between two tasks t_1 and t_2 can only be done when at least one of them has a value: $\mathscr{V}(t_1) = v_1 \vee \mathscr{V}(t_2) = v_2$. When both have a value, the left task is chosen. When none has a value, none can be chosen. These principles lead to the rules **[S-OrLeft]**, **[S-OrRight]**, and **[S-OrNone]**, which encode that the choice operator picks the leftmost task that has a value.

5.5.4 Handling User Input

The handling semantics is the outermost layer of the stack of semantics. It is responsible for performing external steps and choices, and for changing the values of editors. The rules of the interaction semantics are given in fig. 5.13. The semantics is only applicable to normalised tasks t. Interacting with a task, which means sending an input event i to a task t, denoted as $t, \sigma \overset{i}{\Rightarrow} t', \sigma'$, consists of two steps. It first handles the event and then prepares the resulting task for the next input by normalising it.

Inputs i are formed according to the grammar in fig. 5.14. F and S encode the path to the task at which the input is targeted. The observation function $\mathscr{I}$ calculates the possible input events a given task expects. It takes a normalised task and a state and returns a set of inputs that the task can handle. The definition of this function is given in fig. 5.15.

Handling input is done by the *handling* semantics shown in fig. 5.16. It is a small-step semantics with labelled transitions. It takes a task t in a state σ and an input i, and yields a new task t' in a new state σ'.

[H-Change], **[H-Fill]**, **[H-Update]** describe how input events v are used to change the value of editors. Editors only accept values of the correct type.

[H-Next] handles the C (Continue) action, which triggers an external step. As with internal stepping, this is only possible if the left side has a value and the continuation is not failing.

[H-PickLeft], **[H-PickRight]** handle L and R inputs, that are used to pick the left or right option of an external choice.

[H-PassThen], **[H-PassNext]** pass all events other than the continue event C to the left side.

$$\boxed{t, \sigma \mapsto t', \sigma'}$$

[S-Edit]

$$\frac{}{\Box\, v, \sigma \mapsto \Box\, v, \sigma}$$

[S-ThenStay]

$$\frac{t_1, \sigma \mapsto t_1', \sigma' \qquad \mathscr{V}(t_1', \sigma') = \bot}{t_1 \blacktriangleright e_2, \sigma \mapsto t_1' \blacktriangleright e_2, \sigma'}$$

[S-Fill]

$$\frac{}{\boxtimes\, \beta, \sigma \mapsto \boxtimes\, \beta, \sigma}$$

[S-ThenFail]

$$\frac{t_1, \sigma \mapsto t_1', \sigma' \quad \mathscr{V}(t_1', \sigma') = v_1 \quad e_2 v_1, \sigma' \downarrow t_2, \sigma'' \quad \mathscr{F}(t_2, \sigma'')}{t_1 \blacktriangleright e_2, \sigma \mapsto t_1' \blacktriangleright e_2, \sigma'}$$

[S-Update]

$$\frac{}{\blacksquare\, l, \sigma \mapsto \blacksquare\, l, \sigma}$$

[S-ThenCont]

$$\frac{t_1, \sigma \mapsto t_1', \sigma' \quad \mathscr{V}(t_1', \sigma') = v_1 \quad e_2 v_1, \sigma' \downarrow t_2, \sigma'' \quad \neg\mathscr{F}(t_2, \sigma'')}{t_1 \blacktriangleright e_2, \sigma \mapsto t_2, \sigma''}$$

[S-Fail]

$$\frac{}{\lightning, \sigma \mapsto \lightning, \sigma}$$

[S-Xor]

$$\frac{}{e_1 \lozenge e_2, \sigma \mapsto e_1 \lozenge e_2, \sigma}$$

[S-OrLeft]

$$\frac{t_1, \sigma \mapsto t_1', \sigma' \qquad \mathscr{V}(t_1', \sigma') = v_1}{t_1 \blacklozenge t_2, \sigma \mapsto t_1', \sigma'}$$

[S-OrRight]

$$\frac{t_1, \sigma \mapsto t_1', \sigma' \quad \mathscr{V}(t_1', \sigma') = \bot \quad t_2, \sigma' \mapsto t_2', \sigma'' \quad \mathscr{V}(t_2', \sigma'') = v_2}{t_1 \blacklozenge t_2, \sigma \mapsto t_2', \sigma''}$$

[S-OrNone]

$$\frac{t_1, \sigma \mapsto t_1', \sigma' \qquad \mathscr{V}(t_1', \sigma') = \bot \qquad t_2, \sigma' \mapsto t_2', \sigma'' \qquad \mathscr{V}(t_2', \sigma'') = \bot}{t_1 \blacklozenge t_2, \sigma \mapsto t_1' \blacklozenge t_2', \sigma''}$$

[S-Next]

$$\frac{t_1, \sigma \mapsto t_1', \sigma'}{t_1 \rhd e_2, \sigma \mapsto t_1' \rhd e_2, \sigma'}$$

[S-And]

$$\frac{t_1, \sigma \mapsto t_1', \sigma' \qquad t_2, \sigma' \mapsto t_2', \sigma''}{t_1 \bowtie t_2, \sigma \mapsto t_1' \bowtie t_2', \sigma''}$$

Figure 5.12: Striding semantics

$$\boxed{t, \sigma \overset{i}{\Rightarrow} t', \sigma'}$$

[I-Handle]

$$\frac{t, \sigma \overset{i}{\rightarrow} t', \sigma' \qquad t', \sigma' \Downarrow t'', \sigma''}{t, \sigma \overset{i}{\Rightarrow} t'', \sigma''}$$

Figure 5.13: Interaction semantics

$$
\begin{array}{rrcll}
\text{Inputs} & i & ::= & a \mid \mathsf{F}\, i \mid \mathsf{S}\, i & \text{– action, pass to first, pass to second} \\
\text{Actions} & a & ::= & c \mid \mathsf{C} \mid \mathsf{L} \mid \mathsf{R} & \text{– constant, empty, continue, go left, go right}
\end{array}
$$

Figure 5.14: Input grammar

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\mathcal{I} : \text{Task} \times \text{State} \to \mathcal{P}(\text{Input})} \\
\mathcal{I}(\square\, v, \sigma) & = & \{\, c \mid c : \beta \,\} & \textbf{where } \square\, v : \textsc{Task}\ \beta \\
\mathcal{I}(\boxtimes\, \beta, \sigma) & = & \{\, c \mid c : \beta \,\} & \\
\mathcal{I}(\blacksquare\, l, \sigma) & = & \{\, c \mid c : \beta \,\} & \textbf{where } \blacksquare\, l : \textsc{Task}\ \beta \\
\mathcal{I}(\lightning, \sigma) & = & \emptyset & \\
\mathcal{I}(t_1 \blacktriangleright e_2, \sigma) & = & \mathcal{I}(t_1, \sigma) & \\
\mathcal{I}(t_1 \triangleright e_2, \sigma) & = & \mathcal{I}(t_1, \sigma) \cup \{\, \mathsf{C} \mid \mathcal{V}(t_1, \sigma) = v_1 \wedge e_2 v_1, \sigma \Downarrow t_2, \sigma' \wedge \neg \mathcal{F}(t_2, \sigma') \,\} & \\
\mathcal{I}(t_1 \bowtie t_2, \sigma) & = & \{\, \mathsf{F}\, i \mid i \in \mathcal{I}(t_1, \sigma) \,\} \cup \{\, \mathsf{S}\, i \mid i \in \mathcal{I}(t_2, \sigma) \,\} & \\
\mathcal{I}(t_1 \blacklozenge t_2, \sigma) & = & \{\, \mathsf{F}\, i \mid i \in \mathcal{I}(t_1, \sigma) \,\} \cup \{\, \mathsf{S}\, i \mid i \in \mathcal{I}(t_2, \sigma) \,\} & \\
\mathcal{I}(e_1 \lozenge e_2, \sigma) & = & \{\, \mathsf{L} \mid e_1, \sigma \Downarrow t_1, \sigma' \wedge \neg \mathcal{F}(t_1, \sigma') \,\} \cup & \\
 & & \{\, \mathsf{R} \mid e_2, \sigma \Downarrow t_2, \sigma' \wedge \neg \mathcal{F}(t_2, \sigma') \,\} &
\end{array}
$$

Figure 5.15: Inputs

[H-FirstAnd], **[H-SecondAnd]**, **[H-FirstOr]**, **[H-SecondOr]** direct the inputs F and S (First and Second) to the correct branch of parallel combinations.

5.5.5 Implementation

We have implemented the semantics in Haskell. The source code can be found online [Steenvoorden and Naus, 2019]. We use data types as monads [Jaskelioff et al., 2011], data types à la carte [Swierstra, 2008], and monad transformers for layered semantics [Peyton-Jones, 2001]. A command-line interface is part of this implementation. It prompts users to type input events, which are parsed and processed by the interaction semantics. Also, we built an implementation of $\widehat{\text{TOP}}$ combinators on top of iTasks, so that $\widehat{\text{TOP}}$ specifications can be compiled to runnable applications. This shows that $\widehat{\text{TOP}}$ is a subset of iTasks.

5.6 Properties

To show that our semantics are reasonable, we present the following theorems. In this thesis we only state the theorems, the full proofs can be found in the PhD thesis of Naus [2020].

- Evaluation, normalisation and handling semantics are type preserving.
- A progress theorem for the small-step handling semantics.
- The failing function $\mathcal{F}$ only indicates expressions that cannot be normalised and that allow no further interaction.
- The inputs function $\mathcal{I}$ is sound and complete.

$$\boxed{t, \sigma \xrightarrow{i} t', \sigma'}$$

Editing

[H-Change]

$$\frac{v, v' : \beta}{\square\, v, \sigma \xrightarrow{v'} \square\, v', \sigma}$$

[H-Fill]

$$\frac{v : \beta}{\boxtimes\, \beta, \sigma \xrightarrow{v} \square\, v, \sigma}$$

[H-Update]

$$\frac{\sigma(l), v : \beta}{\blacksquare\, l, \sigma \xrightarrow{v} \blacksquare\, l, \sigma[l \mapsto v]}$$

Continuing

[H-Next]

$$\frac{e_2 v_1, \sigma \Downarrow t_2, \sigma' \qquad \mathscr{V}(t_1, \sigma) = v_1 \wedge \neg \mathscr{F}(t_2, \sigma')}{t_1 \rhd e_2, \sigma \xrightarrow{\mathsf{C}} t_2, \sigma'}$$

[H-PickLeft]

$$\frac{e_1, \sigma \Downarrow t_1, \sigma' \qquad \neg \mathscr{F}(t_1, \sigma')}{e_1 \lozenge e_2, \sigma \xrightarrow{\mathsf{L}} t_1, \sigma'}$$

[H-PickRight]

$$\frac{e_2, \sigma \Downarrow t_2, \sigma' \qquad \neg \mathscr{F}(t_2, \sigma')}{e_1 \lozenge e_2, \sigma \xrightarrow{\mathsf{R}} t_2, \sigma'}$$

Passing

[H-PassThen]

$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma'}{t_1 \blacktriangleright e_2, \sigma \xrightarrow{i} t_1' \blacktriangleright e_2, \sigma'}$$

[H-PassNext]

$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma'}{t_1 \rhd e_2, \sigma \xrightarrow{i} t_1' \rhd e_2, \sigma'}$$

[H-FirstAnd]

$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma'}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathsf{F}\,i} t_1' \bowtie t_2, \sigma'}$$

[H-SecondAnd]

$$\frac{t_2, \sigma \xrightarrow{i} t_2', \sigma'}{t_1 \bowtie t_2, \sigma \xrightarrow{\mathsf{S}\,i} t_1 \bowtie t_2', \sigma'}$$

[H-FirstOr]

$$\frac{t_1, \sigma \xrightarrow{i} t_1', \sigma'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathsf{F}\,i} t_1' \blacklozenge t_2, \sigma'}$$

[H-SecondOr]

$$\frac{t_2, \sigma \xrightarrow{i} t_2', \sigma'}{t_1 \blacklozenge t_2, \sigma \xrightarrow{\mathsf{S}\,i} t_1 \blacklozenge t_2', \sigma'}$$

Figure 5.16: Handling semantics

5.6.1 Type Preservation

We say that an expression e is well-typed iff $\Gamma, \Sigma \vdash e : \tau$ for some type τ, and a state is well-typed iff $\Gamma, \Sigma \vdash \sigma$. This means that for all $l \in \sigma$, it holds that $\Gamma, \Sigma \vdash \sigma(l) : \Sigma(l)$.

Theorem 5.6.1 (Type preservation under evaluation)**.** *For all expressions e and states σ such that* $\Gamma, \Sigma \vdash e : \tau$ *and* $\Gamma, \Sigma \vdash \sigma$*, if* $e, \sigma \downarrow e', \sigma'$*, then* $\Gamma, \Sigma \vdash e' : \tau$ *and* $\Gamma, \Sigma \vdash \sigma'$*.* □

Theorem 5.6.2 (Type preservation under normalisation)**.** *For all expressions e and states σ such that* $\Gamma, \Sigma \vdash e : \textsc{Task}\ \tau$ *and* $\Gamma, \Sigma \vdash \sigma$*, if* $e, \sigma \Downarrow e', \sigma'$*, then* $\Gamma, \Sigma \vdash e' : \textsc{Task}\ \tau$ *and* $\Gamma, \Sigma \vdash \sigma'$*.* □

Theorem 5.6.3 (Type preservation under handling)**.** *For all expressions e, states σ and inputs i such that* $\Gamma, \Sigma \vdash e : \textsc{Task}\ \tau$ *and* $\Gamma, \Sigma \vdash \sigma$*, if* $e, \sigma \xrightarrow{i} e', \sigma'$*, then* $\Gamma, \Sigma \vdash e' : \textsc{Task}\ \tau$ *and* $\Gamma, \Sigma \vdash \sigma'$*.* □

All three Theorems are proven to be correct by induction over e. From theorem 5.6.3 and theorem 5.6.2 we directly obtain that the driving semantics also preserves types.

5.6.2 Progress

A well-typed term of type Task is guaranteed to progress after normalisation, unless it is failing. Theorem 5.6.4 formalizes this.

Theorem 5.6.4 (Progress under handling)**.** *For all well-typed expressions e and states σ, if* $e, \sigma \Downarrow e', \sigma'$*, then either* $\mathcal{F}(e', \sigma')$ *or there exist* e''*,* σ''*, and i such that* $e', \sigma' \xrightarrow{i} e'', \sigma''$*.* □

If an expression e and state σ are well-typed, then after normalisation, the pair e', σ' either fails, or there exists some input i that can be handled by it under the handling semantics. To prove this theorem, we need to show that the failing function $\mathcal{F}$ behaves as expected, that is an expression e and state σ are failing if, after normalisation, there exists no input that can be handled by it.

Theorem 5.6.5 (Failing means no interaction possible)**.** *For all expressions e and states σ such that* $\Gamma, \Sigma \vdash e : \textsc{Task}\ \tau$ *and* $\Gamma, \Sigma \vdash \sigma$*, and* $e, \sigma \Downarrow t, \sigma'$*, we have that* $\mathcal{F}(t, \sigma') = \mathsf{True}$*, if and only if there is no input i such that* $t, \sigma' \xrightarrow{i} t', \sigma''$ *for some* t' *and* σ''*.* □

5.6.3 Soundness and Completeness of Inputs

To validate the function $\mathcal{I}$ which calculates all possible inputs, we want to show that the set of possible inputs it produces is both sound and complete with respect to the handling semantics. By sound we mean that all inputs in the set of possible inputs can actually be handled by the handling semantics, and by complete we mean that the set of possible inputs contains all inputs that can be handled by the handling semantics. Theorem 5.6.6 expresses this property.

Theorem 5.6.6 (Inputs function is sound and complete)**.** *For all expressions e, states σ, and inputs i such that* $\Gamma, \Sigma \vdash e : \tau$ *and* $\Sigma \vdash \sigma$*, we have that* $i \in \mathcal{I}(e, \sigma)$ *if and only if there exists an expression* e' *and state* σ' *such that* $e, \sigma \xrightarrow{i} e', \sigma'$*.* □

5.7 Related Work

Our work on TOP lies on the boundary of many areas of study. People have looked at the problem of how to model and coordinate collaboration from many different perspectives. The following subsections give an overview of related work from the many different areas.

5.7.1 Process Algebras

We start the discussion about related work with an in-depth comparison of TOP and Hoare's CSP [Hoare, 1985], as a representative for process algebras. We argue that despite differences in detail, there are striking similarities on a more abstract level. We first summarize the similarities and differences, and then elaborate on them in more detail.

There are some aspects that are similar in TOP and CSP. The first is how internal communication differs from communication with the environment. This difference exists in both TOP and CSP. Internal communication in CSP is introduced with the concealment operator. The semantics of CSP requires that all concealed actions are handled to exhaustion before any action with the environment can take place. This is similar to TOP, where all enabled internal steps must be taken until the system is allowed to react to input events again. In Milner's CCS [Milner, 1989] however, concealed actions are visible to the outside as τ-actions, and can be interleaved with external communication.

Another similarity between TOP and CSP, or any system with concurrency for that matter, is the need for synchronisation. Broadly speaking, concurrency means that different parts of a program can interact with the environment independently, in an interleaved manner. Synchronisation means that only some, but not all, of the possible interleavings are desirable. The semantics of the step combinators in TOP, together with the fact that internal communication happens atomically, allows for concise and intuitive synchronisation code.

There are two main differences between TOP and process algebras. The first is a difference in scope. Process algebras focus on modelling the input/output behaviour of processes, by explicitly stating which actions are sent and received at certain points in the program. The goal of process algebras is formal reasoning about the interaction between processes. Typically, one wishes to prove properties such as deadlock-freedom, liveness, or adherence to a protocol specification. The focus of TOP on the other hand is to model collaboration patterns, with the explicit goal of not having to specify how exactly subtasks communicate. The declarative specification of data dependencies between subtasks enables TOP to hide such details.

The second difference is in the way tasks communicate. TOP knows two forms of communication between tasks: Passing values to continuations and sharing data. This is different from communication in CSP, which is based solely on message-passing.

Communication. Both TOP and CSP have two sorts of communication: with the environment, and between subsystems. In both systems, communication with the *environment* is input and output. They block evaluation of programs until the environment sends or receives events. In both paradigms, communication *between subsystems* does not block. They evaluate programs as far as possible until further external input is required.

Processes communicate with the environment by sending or receiving actions. The CSP process $(a \rightarrow P)$ can engage in action a, after which it continues as process P. Hoare uses the neutral phrase *engaging in an action*, the interpretation of whether an action stands for input

or output is left to the reader. For example, a vending machine $P = (coin \to choc \to P)$ is to be interpreted as taking a coin as input and producing chocolate as output.

Sending and receiving values in CSP is modelled by giving additional structure to the names of actions. For example, an action name could be *in*.5, which can be interpreted as inputting the value 5. In $\widehat{\text{TOP}}$, input values are typed to match the type of the receiving editor, which is enforced by the type- and runtime systems. In CSP action names are essentially strings, subject to implicit restrictions.

CSP uses prefixing $(a \to P)$ for both internal and external communication. Prefixing becomes internal by using *concealment*. The semantics of processes takes any concealed steps as soon as possible, before any further communication with the environment can take place. This means concealed actions have priority over exposed ones. This introduces the possibility of diverging computations, that is processes which continuously take internal steps without ever being able to engage in external communication.

In $\widehat{\text{TOP}}$, external communication is entirely handled by editors. They serve as the means to output values and let users input values. Internal communication is handled by the step combinator and shared values. The step combinator passes the result value of the left-hand task as an argument to the right-hand task, and shared values, allow access to a share from anywhere in the program.

Concurrency. Both tasks and processes are models of *multiprogramming*, as opposed to *multiprocessing*. This means that one processor evaluates multiple programs in an interleaving fashion. Multiprogramming involves three aspects: concurrency, synchronisation, and nondeterminism.

In CSP, there are two different combinators for parallel composition: parallel and interleave. The parallel process $P \parallel Q$ can take P-steps and Q-steps in arbitrary interleaving for actions unique to P and Q. Actions in the alphabets of both P and Q must be taken in synchronisation, so if one process wants to take such a step, it must wait until the other is ready to do so. Synchronised steps are the only occasion where two processes actually do something at the same time. The interleaving operator $P \mid\mid\mid Q$ does not synchronise any actions of P and Q. All interleavings are permitted. Actions that occur in both alphabets are nondeterministically taken by either P or Q.

Synchronisation. Synchronisation means that an agent has to pause execution until some condition is met. The need for synchronisation arises quite naturally in situations involving concurrency. For multiprogramming, a *synchronisation problem* always means that some of the possible interleavings are forbidden. For example, the mutual exclusion problem for two parallel processes

$$a_p \to b_p \to \text{STOP}$$
$$a_q \to b_q \to \text{STOP}$$

can be stated as: In no interleaving should there be two adjacent as. The intuition is that a and b mark the beginning and end of a critical section, in which no other task should be able to intrude. The mutual exclusion problem for two parallel tasks can be stated as follows. Let inc and dec be tasks that increment and decrement some shared counter, initialized to 0. Then, while executing the program:

$$(\Box\ \langle\rangle \rhd \mathsf{inc}\ x \rhd \mathsf{dec}\ x) \bowtie (\Box\ \langle\rangle \rhd \mathsf{inc}\ x \rhd \mathsf{dec}\ x)$$

the shared counter should at no point be greater than 1. This means that in no interleaving should there be two adjacent incs.

To solve synchronization problems, programmers must amend the problematic programs, using the synchronization primitives of the language, so that the forbidden interleavings cannot occur. The basic method for synchronisation in CSP is synchronised prefixing. If some action a is in the alphabets of both P and Q, then their parallel composition $P \parallel Q$ can take a-steps only if both P and Q are ready to take an a-step, in which case they both take this step simultaneously. Both have to wait for the condition that the other process is ready to take the step. All synchronisation problems in CSP must be solved by employing synchronised steps in some form. The basic synchronization method in $\widehat{\text{TOP}}$ is guarded stepping. A step is guarded, which means it cannot be taken, if its right-hand side evaluates to $\lightning$. By using conditionals and shared values, parallel tasks can guard and un-guard each others steps. For example, the following program cannot proceed until some other parallel task increases the shared counter s.

$$\blacksquare s \rhd \lambda c\ .\ \textbf{if}\ c < 7\ \textbf{then}\ \lightning\ \textbf{else}\ \ldots$$

Semaphores. Semaphores are a synchronization mechanism on a medium level of abstraction. They usually do not exist as primitives in a language, but as a library that is itself implemented using the language's primitive synchronization features. Semaphores have two operations *signal* and *wait*, which can be used in different ways to solve a variety of synchronization problems [Downey, 2008]. Semaphores are supposed to be shared among agents. Semaphores have an internal counter and two methods *signal* and *wait*. The counter can be initialized once, but subsequently not accessed directly. When an agent invokes *wait* and the counter is greater than zero, the semaphore decreases the counter and the agent continues. When an agent invokes *wait* while the counter is zero, the agent blocks and has to wait until the counter increases. When an agent invokes *signal*, the counter is increased. If there are any agents waiting for the semaphore, one of them is woken up.

In CSP, a semaphore is a process that runs in parallel with all processes it is supposed to coordinate. For the sake of this example, we study a semaphore with maximum counter of 2, initialized to 1, that coordinates two processes P and Q. The example can be generalized to an arbitrary counter and arbitrary many processes.

$$\begin{aligned}
Sem &= C_1 \\
C_2 &= (wait_p \to C_1 \mid wait_q \to C_1) \\
C_1 &= (wait_p \to C_0 \mid wait_q \to C_0 \\
&\qquad \mid signal_p \to C_2 \mid signal_q \to C_2) \\
C_0 &= (signal_p \to C_1 \mid signal_q \to C_1)
\end{aligned}$$

The semaphore starts in state C_1, which represents a counter of 1. Every process has its own actions *signal* and *wait*, because we want these actions to be synchronized only between semaphore and process, not directly between the processes. In state C_1, the semaphore can engage in *wait* and *signal* actions for both processes. *wait* decreases the counter, *signal* increases the counter. In state C_0, which represents a counter of 0, only *signal* actions are

possible. A thread willing to engage in *wait* while the semaphore is in C_0 must wait until the other process has engaged in *signal* with the semaphore.

Using such semaphores, the mutual exclusion problem in CSP can be solved with the following program. The semaphore and processes P and Q are run in parallel, and P and Q must wait for the semaphore before entering the critical section. After leaving the critical section, they signal the semaphore.

$$\begin{aligned} \mathit{Sem} \parallel \mathit{wait}_p \to a_p \to b_p \to \mathit{signal}_p \to \mathrm{STOP} \\ \parallel \mathit{wait}_q \to a_q \to b_q \to \mathit{signal}_q \to \mathrm{STOP} \end{aligned}$$

In $\widehat{\text{TOP}}$, a semaphore is a shared integer, together with two tasks *signal* and *wait*.

$$\begin{aligned} &\text{type Semaphore} = \textsc{Ref Int} \\ &\mathit{signal} :: \text{Semaphore} \to \textsc{Task Unit} \\ &\mathit{wait} :: \text{Semaphore} \to \textsc{Task Unit} \end{aligned}$$

The function *signal* simply increments the shared counter. The function *wait* watches the counter and has a guarded step that is enabled only if the counter is greater than zero. The editor $\blacksquare\, s$ should be regarded as read-only.

$$\begin{aligned} \mathit{signal}\ s &= s := !s + 1; \square\langle\rangle \\ \mathit{wait}\ s &= \blacksquare\, s \rhd \lambda c.\ \textbf{if}\ c > 0\ \textbf{then}\ s := c - 1; \square\langle\rangle\ \textbf{else}\ \lightning \end{aligned}$$

When the step in *wait* is enabled and receives a continue-event, it decrements the counter. Decrementing the counter happens in the normalization semantics, and is therefore atomic with respect to event handling. When this causes the counter to become zero, all tasks that happen to wait for the semaphore have their step disabled simultaneously. Once the semaphore gets signalled, all steps become enabled. The first task to take the step again disables the step for all others.

With semaphores like this, the mutual exclusion problem in $\widehat{\text{TOP}}$ can be solved with the following program. Let s be a shared integer.

$$\begin{aligned} &(\text{wait } s \rhd \text{inc } x \rhd \text{dec } x \rhd \text{signal } s) \bowtie \\ &(\text{wait } s \rhd \text{inc } x \rhd \text{dec } x \rhd \text{signal } s) \end{aligned}$$

This demonstrates that with a parallel combinator comes the need for synchronization, in both CSP and $\widehat{\text{TOP}}$. While the synchronization primitives in both systems are different, they can be used to implement semaphores, which can be used to solve the mutual exclusion problem in a quite similar manner.

Nondeterminism. Nondeterminism means that a system can react to the same input in different ways, the choice of which cannot be influenced by the environment. CSP has an operator to explicitly introduce nondeterminism. Furthermore, nondeterminism can arise from the combination of some other combinators, for example choice and concealment. Hoare states that nondeterminism is only useful for *specifying* processes, never for *implementing* them [Hoare, 1985]. CSP can be used for both. A process with nondeterministic behaviour must always be regarded as a specification, while a deterministic process can be seen as specification or implementation. $\widehat{\text{TOP}}$ does not have nondeterminism. For every accepted input, a $\widehat{\text{TOP}}$ program can only react in one way.

5.7.2 TOP Implementations

iTasks. As mentioned earlier, iTasks is an implementation of TOP. iTasks has many features, and its basic combinators are versatile and powerful. Simpler combinators are implemented by restricting the powerful ones. This is useful for everyday programming, where having lots of functionality at one's fingertips is convenient and efficient. $\widehat{\text{TOP}}$ on the other hand does not include the many different variations of the step- and parallel combinators of iTasks. To name two examples, the combinators `(>>|)` and `(||-)` are variations of step and parallel that ignore the value of the left task.

There have been two previous papers that describe semantics of iTasks, by Koopman et al. [2008] and Plasmeijer et al. [2012]. Both present semantics in the form of minimal implementations of a subset of the interface of iTasks. These semantics do not make an explicit distinction between the host language and task language and they do not provide an explicit formal semantics. We describe the $\widehat{\text{TOP}}$ semantics by giving semantic relations and an implementation, which makes our system better suited for formal reasoning. When comparing the features offered by $\widehat{\text{TOP}}$ to iTasks, we see that the task layer of both languages are very similar. One important distinction is that iTasks has notions of time and task stability, which are not present in $\widehat{\text{TOP}}$. Nevertheless, we expect that $\widehat{\text{TOP}}$ can emulate the behaviour of the task layer of iTasks, but we leave proving this as future work.

mTasks. The mTasks framework [Koopman et al., 2018] is an implementation of TOP geared towards IOT devices. As in $\widehat{\text{TOP}}$, its basic combinators are a subset of iTasks. However, on IOT devices it is useful to continue running tasks indefinitely, which is done in mTasks using a `forever` combinator. This is currently not possible in $\widehat{\text{TOP}}$. As for iTasks, there is currently no formal semantics for mTasks.

5.7.3 Worfklow Modelling

A lot of research has been done into workflow modelling. The systems described in the literature mostly follow a *boxes and arrows* model of specifying workflows. Control flow, represented by arrows, usually can go unrestricted from anywhere to anywhere else in a workflow. TOP on the other hand specifies workflows declaratively, avoiding explicit specification of control flow.

Workflow patterns. Workflow patterns are to workflows what design patterns are to software engineering. They are recurring patterns in workflow systems, much like the combinators defined in $\widehat{\text{TOP}}$. Work by van der Aalst et al. [2003] defines a long list of such pattens, and examines their availability in industry workflow software. Workflow patterns are usually described in terms of control flow graphs, and no formal specification is given, which makes comparison and formal reasoning difficult.

Workflow Nets & YAWL. Workflow Nets (WFN) [van der Aalst, 1998] allow for the modelling and analysis of business processes. They are graphical in nature, and display how components are related to each other. A downside of WFNs is that they do not facilitate higher-order constructs, and they are not directly executable. YAWL is a language based on WFN that is directly executable [van der Aalst and ter Hofstede, 2005]. It allows modelling and execution of dynamic workflows, with support for *and*, *or* and *xor* workflow patterns.

BPEL. BPEL [OASIS, 2019] is another popular business process calculus. It is a standardised language that allows for the specification of actions within business processes using an XML format. The language is mainly used for coordinating web services. Two workflow patterns are supported, execution of services can be done sequentially or in parallel. On top of that, processes can be guarded by conditions. There is no support for higher-order processes. Processes described in BPEL can be regarded as activity graphs, and they can also be rendered as such. The specified processes in BPEL are directly executable, just like YAWL.

5.7.4 Reactive Programming

HipHop & Esterel. HipHop [Berry et al., 2011; Berry and Serrano, 2013] is a programming language tailored to the development of synchronous reactive web systems. From a single source, both server and client applications can be generated. Programs are written in the Hop language, a Scheme dialect. Communication is based on a reactive layer embedded in Hop. The set of HipHop reactive statements is based on those of the Esterel language [Boussinot and De Simone, 1991; Berry and Gonthier, 1992]. Each reactive component starts by specifying possible input and output events. The component then proceeds as a state machine. Input events are sent to such a machine programmatically using Hop, or are explicitly wired to events from the client. They are optionally associated with a Hop value. As Hop is a dynamic language, and HipHop uses strings to identify events, events and their possible associated values are not statically checked. Events are aggregated until the moment the machine is asked to react. The machine is executed and reacts by building a multi-set of output events. The execution of a HipHop machine is atomic. The set of inputs is not influenced by the current computations.

As with $\widehat{\text{TOP}}$, HipHop is a DSL embedded in a general-purpose programming language. Another similarity is that both specifications lead to executable server and client applications from a single source. However, both HipHop and Esterel are more low-level regarding their specification. Where $\widehat{\text{TOP}}$ takes tasks and collaboration as a starting point, HipHop focusses on synchronous communication and atomic execution of reactive machines. This difference in focus can be seen in the way both systems define events. In HipHop programmers can define and use their own events. Inputs in $\widehat{\text{TOP}}$ are not extensible and not visible to the developer. They are a completely separate entity living on the semantic level. Another important difference is the way in which both systems handle events. In HipHop the programmer decides when a machine should process its events. This could be just one event, or a multi-set of events that are processed simultaneously. $\widehat{\text{TOP}}$ always processes an input the moment it occurs and only handles a single event in one instance.

Functional reactive programming. The Functional Reactive Programming (FRP) paradigm allows programmers to describe dynamic changes of values in a declarative way. This is done by specifying networks of values, called behaviours, that can depend on each other and on external events. Behaviours can change over time, or triggered by events. When a behaviour changes, all other behaviours that depend on it are updated automatically. The underlying implementation that takes care of the updating usually can tie input devices, like mouse and keyboard, to event streams and behaviours to output facilities, like text fields. This allows for declarative specifications of applications with user interfaces. The idea of FRP was pioneered by Elliott and Hudak [1997]. Since then many variants have

been developed, the most well-known of which include reactive-banana [Apfelmus, 2019], FrTime [Cooper and Krishnamurthi, 2004], and Flapjax [Meyerovich et al., 2009].

FRP and TOP are different systems with different goals. Whereas FRP expresses automatically updating data dependencies, TOP expresses collaboration patterns. TOP has no notion of time. Tasks cannot change spontaneously over time, while behaviours can. Only input events can change task values. The biggest conceptual difference between a workflow in TOP and a data network in FRP is that an event to a task only causes updates up until the next step, while an event in FRP propagates through the whole network.

There are some concepts that are similar in TOP and FRP. The *stepper* behaviour, for example, is associated with an event and yields the value of the most recent event. This is similar to editors in TOP. Furthermore, both systems can be used to declaratively program user interfaces, albeit in FRP the programmer has to construct the GUI elements manually, and connect inputs and outputs to the correct events and behaviours. In TOP graphical user interfaces are automatically derived.

5.7.5 Session Types

Session types are a type discipline that can be used to check whether communicating programs conform to a certain protocol. Session types are expressions in some process calculus that describe the input/output behaviour of their programs. Session types are useful for programming languages where modules communicate with each other via messages, like CSP, π-calculus, or Go, to name a few. The only form of messages in TOP are input events which drive execution, but modules do not communicate using messages. Therefore, session types are not relevant for TOP in the sense used in the literature. Formal reasoning about TOP programs is one of our future goals for $\widehat{\text{TOP}}$. The ideas and techniques of session types could be useful for specifying that a list of inputs of a certain form leads to desired task values. The details are a topic for future work.

5.8 Conclusion

In this chapter we have identified and intuitively described the essence of task-oriented programming. We then formalised this essence by developing a domain-specific language for declarative interactive workflows, called $\widehat{\text{TOP}}$. The task language and the host language are clearly separated, to make explicit where the boundaries are. The semantics of the task layer is driven by user input. We have compared $\widehat{\text{TOP}}$ with workflow modelling languages, process algebras, functional reactive programming and session types to point out differences and similarities. Finally, we have proven type safety and progress for our language.

Future work. There are a couple of ways in which we would like to continue this line of work. One of the main motivations to formalise task-oriented programming is to be able to reason about programs. In this chapter we reason about the language itself, but it would be nice to prove properties about individual programs. To this end, we are very interested to see if it is possible to develop an axiomatic semantics for $\widehat{\text{TOP}}$ that allows us to do so. There are certain properties of our language that make this particularly complex: We have to deal with parallelism, user interaction, and references.

We would also like to prove whether certain programs are equivalent, for example to show that the monad laws hold for the step combinator. This requires a notion of equality, which in the presence of side effects most certainly needs some form of coalgebraic input-output conformance. We have implemented the reduction semantics of our language in Haskell, whose type system could aid in the formalisation of such proofs.

Another form of reasoning about programs is static analysis. Klinik et al. [2017b] have developed a cost analysis for tasks that require resources. This analysis was developed for a simpler task language, and could be brought over to the one developed here.

Naus and Jeuring [2016] have looked at building a generic feedback system for rule-based problems. A workflow system typically is rule-based, as outlined in their work. It would be interesting to fit the generic feedback system to $\widehat{\text{TOP}}$, to support end users working in applications developed in this language.

Additionally, we would like to develop visualisations for $\widehat{\text{TOP}}$ language constructs. An assistive development environment integrating these visualisations and the presented textual language would aid domain experts to model workflows in a more accessible manner. A similar system for iTasks has been developed in the past by Stutterheim et al. [2014].

6 A Symbolic Execution Semantics for TopHat

In this chapter we aim to make formal verification of software written in TOP easier. Currently, only testing is used to verify that programs behave as intended. We employ symbolic execution to guarantee that no aberrant behaviour can occur. In the previous chapter we presented TopHat, a formal language that implements the core aspects of TOP. In this chapter we develop a symbolic execution semantics for TopHat. Symbolic execution allows to prove that a given property holds for all possible execution paths of TopHat programs. We show that the symbolic execution semantics is consistent with the original TopHat semantics, by proving soundness and completeness. We present an implementation of the symbolic execution semantics in Haskell. By running example programs, we validate our approach. This chapter represents a step forward in the formal verification of TOP software.

6.1 Introduction

The usefulness of TOP has been demonstrated in several projects that used it to implement various applications. It has been used by the Royal Netherlands Navy [Klinik et al., 2018], the Dutch tax office [Stutterheim et al., 2017] and the Dutch coast guard [Lijnse et al., 2012]. Furthermore, it could be applied in domains like healthcare and Internet of Things [Koopman et al., 2018]. Applications in these domains are often mission critical, where programming errors can have severe consequences. In the previous chapter, we presented the programming language $\widehat{\text{TOP}}$, to distill the core features of TOP into a language suitable for formal treatment. To verify that a $\widehat{\text{TOP}}$ program behaves as intended, we would like to show that it satisfies given properties. A common way to do this is to write test cases, or to generate random input, and verify that all outcomes satisfy the property. Writing tests manually is time consuming and cumbersome. Testing interactive applications needs people to operate the application, maybe making use of a way to record and replay interactions. With this kind of testing there is no guarantee that all possible execution paths are covered.

To overcome these issues, we employ symbolic execution. Instead of executing tasks with test input, we run tasks on symbolic input. Symbolic input consists of tokens that represent any value of a certain type. When a program branches, the execution engine records the conditions over the symbolic input that lead to the different branches. These conditions can then be compared to given predicates to check if the predicates hold under all conditions. We let an SMT solver verify these predicates. In this way we can guarantee that given predicates over the outcome of a TOP program always hold. Since iTasks is too big for formal reasoning, we apply symbolic execution to $\widehat{\text{TOP}}$, by systematically changing the semantic rules of the original language.

6.1.1 Contributions

In this chapter we make the following contributions.

- We present a symbolic execution semantics for $\widehat{\text{TOP}}$, a programming language for workflows embedded in the simply-typed lambda calculus.
- We prove soundness and completeness of the symbolic semantics with respect to the original $\widehat{\text{TOP}}$ semantics.
- We present an implementation of the symbolic execution semantics in Haskell.

6.1.2 Structure

Section 6.2 gives a brief overview of $\widehat{\text{TOP}}$ and its concepts. Section 6.3 introduces some examples to demonstrate the goal of our symbolic execution analysis. In section 6.4, the $\widehat{\text{TOP}}$ language is defined. Section 6.5 goes on to define the formal semantics of the symbolic execution. In section 6.6, soundness and completeness are shown for the symbolic execution semantics with respect to the original $\widehat{\text{TOP}}$ semantics. In section 6.7 related work is discussed, and section 6.8 concludes.

6.2 TopHat Recapitulation

This section briefly summarizes the task-oriented programming language $\widehat{\text{TOP}}$, and discusses our vision about symbolic evaluation of this language. $\widehat{\text{TOP}}$ consists of two parts, the host language and the task language. Programs in $\widehat{\text{TOP}}$ are called *tasks*. The basic elements of tasks are editors. Using combinators, tasks can be combined into larger tasks. The task language is embedded in a simply-typed lambda calculus with references, conditionals, booleans, integers, strings, pairs, lists and unary and binary operations on these types. References allow tasks to communicate with each other, sharing information across task boundaries. The simply-typed lambda calculus does not have recursion. By restricting references to only hold basic types, strong normalisation of the calculus is guaranteed. The full syntax of the host language is listed in section 6.4. Next, we discuss the main constructs of the task language.

6.2.1 Editors

Editors are the most basic tasks. They are used to communicate with the outside world. Editors are an abstraction over widgets in a GUI library or on webpage forms. Users can change the value held by an editor, in the same way they can manipulate widgets in a GUI. When a TOP implementation generates an application from a task specification, it derives user interfaces for the editors. The appearance of an editor is influenced by its type. For example, an editor for a string can be represented by a simple input field, a date by a calendar, and a location by a pin on a map. There are three different editors in $\widehat{\text{TOP}}$.

$\square\, v$ Valued editor. This editor holds a value v of a certain type. The user can replace the value by a new value of the same type.

$\boxtimes\, \tau$ Unvalued editor. This editor holds no value, and can receive a value of type τ. When that happens, it turns into a valued editor.

■ l Shared editor. This editor refers to a store location l. Its observable value is the value at that location. When it receives a new value, the value is stored at location l.

6.2.2 Combinators

Editors can be combined into larger tasks using combinators. Combinators describe the way people collaborate. Tasks can be performed in sequence or in parallel, or there is a choice between two tasks. The following combinators are available in $\widehat{\text{TOP}}$. Here, t stands for tasks and e for arbitrary expressions.

$t \blacktriangleright e$ Step. Users can work on task t. As soon as t has a value, that value is passed on to the right-hand side e. The expression e is a function, taking the value as an argument, resulting in a new task.

$t \triangleright e$ User Step. Users can work on task t. When t has a value, the step becomes enabled. Users can then send a continue event to the combinator. When that happens, the value of t is passed to the right-hand side, with which it continues.

$t_1 \bowtie t_2$ Composition. Users can work on tasks t_1 and t_2 in parallel.

$t_1 \blacklozenge t_2$ Choice. The system chooses between t_1 or t_2, based on which task first has a value. If both tasks have a value, the system chooses the left one.

$e_1 \lozenge e_2$ User choice. A user has to make a choice between either the left or the right-hand side. The user continues to work on the chosen task.

In addition to editors and combinators, $\widehat{\text{TOP}}$ also has the task *fail* ($\lightning$). Programmers can use this task to indicate that a task is not reachable or viable. When the right-hand side of a step combinator evaluates to $\lightning$, the step will not proceed to that task.

6.2.3 Observations

Several observations can be made on tasks. The value function $\mathscr{V}$ retrieves the current value of a task. The value function is a partial function, since not all tasks have a value, for example empty editors and steps. One can also observe whether a task is failing, by means of the failing function $\mathscr{F}$. The task $\lightning$ is failing, as is a parallel combination of failing tasks ($\lightning \bowtie \lightning$). The step combinator makes use of both functions in order to determine if it can step. First, it uses $\mathscr{V}$ to see if the left-hand side produces a value. If that is the case, it uses the $\mathscr{F}$ function to see if it is safe to step to the right-hand side. The complete definition of the value and failing function are discussed in section 6.5.2.

6.2.4 Input

Input events drive evaluation of tasks. Tasks are typed, input is typed as well, and editors only accept input of the correct type. Examples of input events are replacing a value in an editor, or sending a continue event to a user step. When the system receives a valid event, it hands this event to the current task, which reduces to a new task. Everything in between interaction steps is evaluated atomically with respect to inputs. Input events are synchronous, which means that the order of execution is completely determined by the order of the events. In particular, the order of input events determines the progression of parallel branches.

6.3 Examples

In this section we study three examples to illustrate what kind of properties of task-oriented programs we would like to prove.

6.3.1 Positive Value

This example demonstrates how we can prove that the first observable value of a program can only be a positive number. Consider the program in fig. 6.1. It asks the user to input a value of type Int. This value is then passed on to the right-hand side. If the value is greater than zero, an editor containing the entered value is returned. At this point, the task has an observable value, and we consider it done. Otherwise the step does not proceed and the task does not have an observable value. The user can enter a different input value.

Symbolic execution of this program proceeds as follows. The symbolic execution engine generates a fresh symbolic input s for the editor on the left. The engine then arrives at the conditional. To take the then-branch, the condition $s > 0$ needs to hold. This branch will then result in $\Box\, s$, in which case the program has an observable value. The engine records this endpoint together with its path constraint $s > 0$. The else-branch gets executed if the condition does not hold, but this leads to a failing task. Therefore, the step is not taken and the task expression is not altered. No additional program states are generated. Symbolic execution returns a list of all possible program end states, together with the path constraints that led to them. If all end states satisfy the desired property, it is guaranteed that the property holds for all possible inputs. In this example, the only end state is the expression $\Box\, s$ with path constraint $s > 0$. From that we can conclude that no matter what input is given, the only result value possible is greater than zero.

6.3.2 Tax Subsidy Request

Stutterheim et al. [2017] worked with the Dutch tax office to develop a demonstrator for a fictional but realistic law about solar panel subsidies. In this section we study a simplified version of this, translated to $\widehat{\text{TOP}}$, to illustrate how symbolic execution can be used to prove that the program implements the law. This example proves that a citizen will get subsidy only under the following conditions.

- The roofing company has confirmed that they installed solar panels for the citizen.
- The tax officer has approved the request.
- The tax officer can only approve the request if the roofing company has confirmed, and the request is filed within one year of the invoice date.
- The amount of the granted subsidy is at most 600 EUR.

$$\boxtimes \text{Int} \blacktriangleright \lambda x.\ \textbf{if}\ x > 0\ \textbf{then}\ \Box x\ \textbf{else}\ \lightning$$

Figure 6.1: A task that only steps on a positive input value

```
let today = 18 September 2020 in                                                    1
let provideDocuments = ⊠Amount ⋈ ⊠Date in                                           2
let companyConfirm = □True ♢ □False in                                              3
let officerApprove = λinvoiceDate. λdate. λconfirmed.                               4
  □False ♢ if (date − invoiceDate < 365 ∧ confirmed) then □True else ↯ in           5
provideDocuments ⋈ companyConfirm ▶ λ⟨⟨invoiceAmount, invoiceDate⟩ , confirmed⟩ .  6
officerApprove invoiceDate today confirmed ▶ λapproved.                             7
let subsidyAmount = if approved then min 600 (invoiceAmount / 10) else 0 in         8
□⟨subsidyAmount, approved, confirmed, invoiceDate, today⟩                           9
```

Figure 6.2: Subsidy request and approval workflow at the Dutch tax office

Figure 6.3: Graphical user interface for the task in fig. 6.2

Figure 6.2 shows the program. To enhance readability of the example, we omit type annotations and make use of pattern matching on tuples. A date is specified using an integer representing the number of days since 1 January 2000.

The program works as follows. First (line 6) in parallel the citizen has to provide the invoice documents of the solar panels, while the roofing company has to confirm that they have actually installed solar panels at the citizen's address. Once the invoice and the confirmation are there, the tax officer has to approve the request (line 7). The officer can always decline the request, but they can only approve it if the roofing company has confirmed and the application date is within one year of the invoice date (line 5). The result of the program is the subsidy amount, together with all information needed to prove the required properties (line 9). The program's graphical user interface is shown in fig. 6.3.

The result of the overall task is a tuple with the subsidy amount, the officer's approval, the roofing company's confirmation, the invoice date, and today's date. Returning all this information allows the following predicate to be stated. The predicate has 5 free variables, which correspond to the returned values.

$$\psi(s,a,c,i,t) = s > 0 \implies c \tag{6.1}$$
$$\wedge\ s > 0 \implies a \tag{6.2}$$
$$\wedge\ a \implies (c \wedge t - i < 365) \tag{6.3}$$
$$\wedge\ s \leq 600 \tag{6.4}$$
$$\wedge\ \neg a \implies s \equiv 0 \tag{6.5}$$

The predicate ψ states that (6.1) if subsidy s has been payed, the roofing company must have confirmed c, (6.2) if subsidy has been payed, the officer must have approved a, (6.3) the officer can approve only if the roofing company has confirmed and today's date t is

```
let maxSeats = 50 in                                                    1
let bookedSeats = ref [] in                                             2
let bookSeat = ⊠INT ▶ λx .                                              3
  if not (x ∈ !bookedSeats) ∧ x ≤ maxSeats                              4
    then bookedSeats := x :: !bookedSeats ▶ λ_ . □x                     5
    else ↯ in                                                           6
bookSeat ⋈ bookSeat ⋈ bookSeat ▶ λ_ .                                   7
□(!bookedSeats)                                                         8
```

Figure 6.4: Flight booking

within 356 days of the invoice date i, and (6.4) the subsidy is maximal 600 EUR. Finally, (6.5) if the officer has not approved, the subsidy must be 0. Our system is able to prove this property about the program in fig. 6.3.

6.3.3 Flight Booking

Recall the flight booking system from section 5.2. We prove that when the program terminates, every passenger has exactly one seat, and that no two passengers have the same seat. The example program listed in fig. 6.4 is a simplified version of what we presented in section 5.2. The program consists of three parallel seat booking tasks (line 7). There is a shared list that stores all booked seats so far (line 2). To book a seat, a passenger has to enter a seat number (line 3). A guard expression makes sure that only free seats can be booked (line 4). The exclamation mark denotes dereferencing. When the guard is satisfied, the list of booked seats is updated, and the user can see his booked seat (line 5). The main expression runs the seat booking task three times in parallel (line 7), simulating three concurrent customers. The program returns the list of booked seats. With the returned list, we can state the predicate to verify the correctness of the booking process.

$$\psi(l) = \mathsf{len}\, l \equiv 3 \tag{6.1}$$
$$\wedge\ \mathsf{uniq}\, l \tag{6.2}$$

The predicate specifies that all three passengers booked exactly one seat (6.1), and that all seats are unique (6.2), which means that no two passengers booked the same seat. The unary operators for list length (len) and uniqueness (uniq) are available in the predicate language. List length is a capability of SMT-LIB, while uniq is our own addition.

6.4 Language

The language presented in this section, Symbolic $\widehat{\text{TOP}}$, is nearly identical to $\widehat{\text{TOP}}$ of chapter 5. The main difference with the original grammar is the addition of symbolic values. Symbolic execution for functional programming languages struggles with higher-order features. This topic is under active study [Hallahan et al., 2017, 2019], and is not the focus of our work. Therefore, we restrict symbols to only represent values of basic types. This restriction is of little importance in the domains we are interested in. Allowing users to enter higher-order

Symbolic expressions

$$
\begin{array}{lll}
\tilde{e} ::= & \lambda x : \tau.\, \tilde{e} \mid \tilde{e}_1\ \tilde{e}_2 \mid x & \text{– abstraction, application, variable} \\
\mid & c \mid \langle\rangle \mid u\ \tilde{e}_1 \mid \tilde{e}_1\ o\ \tilde{e}_2 & \text{– constant, unit, unary, binary operation} \\
\mid & \textbf{if}\ \tilde{e}_1\ \textbf{then}\ \tilde{e}_2\ \textbf{else}\ \tilde{e}_3 & \text{– conditional} \\
\mid & \langle \tilde{e}_1, \tilde{e}_2 \rangle \mid \mathsf{fst}\ \tilde{e} \mid \mathsf{snd}\ \tilde{e} & \text{– pair, projections} \\
\mid & [\,]_\beta \mid \tilde{e}_1 :: \tilde{e}_2 & \text{– nil, cons} \\
\mid & \mathsf{head}\ \tilde{e} \mid \mathsf{tail}\ \tilde{e} & \text{– first element, list tail} \\
\mid & \textbf{ref}\ \tilde{e} \mid !\tilde{e} \mid \tilde{e}_1 := \tilde{e}_2 \mid l & \text{– references, location} \\
\mid & \tilde{p} \mid s & \text{– symbolic pretask, symbol}
\end{array}
$$

Symbolic Pretasks

$$
\begin{array}{lll}
\tilde{p} ::= & \square\,\tilde{e} \mid \boxtimes\,\beta \mid \blacksquare\,\tilde{e} & \text{– valued editor, unvalued editor, shared editor} \\
\mid & \tilde{e}_1 \blacktriangleright \tilde{e}_2 \mid \tilde{e}_1 \triangleright \tilde{e}_2 & \text{– internal step, external step} \\
\mid & \lightning \mid \tilde{e}_1 \bowtie \tilde{e}_2 & \text{– fail, parallel composition} \\
\mid & \tilde{e}_1 \blacklozenge \tilde{e}_2 \mid \tilde{e}_1 \lozenge \tilde{e}_2 & \text{– internal choice, external choice}
\end{array}
$$

Figure 6.5: Syntax of Symbolic $\widehat{\text{TOP}}$ expressions

Values

$$
\begin{array}{lll}
\tilde{v} ::= & \lambda x : \tau.\, \tilde{e} \mid \langle \tilde{v}_1, \tilde{v}_2 \rangle \mid \langle\rangle \mid c & \text{– abstraction, pair, unit, constant} \\
\mid & [\,]_\beta \mid \tilde{v}_1 :: \tilde{v}_2 \mid l \mid \tilde{t} & \text{– nil, cons, location, task} \\
\mid & u\ \tilde{v} \mid \tilde{v}_1\ o\ \tilde{v}_2 \mid s & \text{– unary/binary operation, symbol}
\end{array}
$$

Tasks

$$
\begin{array}{lll}
\tilde{t} ::= & \square\,\tilde{v} \mid \boxtimes\,\beta \mid \blacksquare\,l & \text{– valued editor, unvalued editor, shared editor} \\
\mid & \tilde{t}_1 \blacktriangleright \tilde{e}_2 \mid \tilde{t}_1 \triangleright \tilde{e}_2 & \text{– internal step, external step} \\
\mid & \lightning \mid \tilde{t}_1 \bowtie \tilde{t}_2 & \text{– fail, parallel combination} \\
\mid & \tilde{t}_1 \blacklozenge \tilde{t}_2 \mid \tilde{e}_1 \lozenge \tilde{e}_2 & \text{– internal choice, external choice}
\end{array}
$$

Figure 6.6: Syntax of values in Symbolic $\widehat{\text{TOP}}$

values is not useful in most workflow applications. By restricting the input grammar to first-order values only, we ensure that no higher-order user input can be entered. Apart from input, all other higher-order features are unrestricted. The following subsections describe in detail how all elements of the $\widehat{\text{TOP}}$ language deal with the addition of symbols.

6.4.1 Expressions, Values, and Types

The syntax of Symbolic $\widehat{\text{TOP}}$ is listed in fig. 6.5. Two main changes have been made with regards to the original $\widehat{\text{TOP}}$ grammar. First, symbols s have been added to the syntax of expressions. However, they are not intended to be used by programmers, similar to locations l. Instead, they are generated by the semantics as placeholders for symbolic inputs. Symbols are treated as values (fig. 6.6). They have therefore been added to the grammar of values. Also, every symbol has a type, and basic operations can take symbols as arguments.

[T-Sym]

$$\frac{s : \beta \in \Gamma}{\Gamma, \Sigma \vdash s : \beta}$$

Figure 6.7: Additional typing rule for Symbolic $\widehat{\text{TOP}}$

Symbolic inputs

$\tilde{\imath}$ ::= $\tilde{a}$ | $\mathsf{F}\,\tilde{\imath}$ | $\mathsf{S}\,\tilde{\imath}$ – symbolic action, to first, to second

Symbolic actions

$\tilde{a}$ ::= s | C | L | R – symbol, continue, go left, go right

Figure 6.8: Syntax of inputs and actions in Symbolic $\widehat{\text{TOP}}$

Path constraints

φ	::=	$c \mid s \mid \langle\rangle \mid \langle \varphi_1, \varphi_2 \rangle$	– constant, symbol, unit, pairs
	|	$[\,]_\beta \mid \varphi_1 :: \varphi_2$	– nil, cons
	|	$u\ \varphi \mid \varphi_1\ o\ \varphi_2$	- unary operation, binary operation

Figure 6.9: Syntax of path constraints

The types of Symbolic $\widehat{\text{TOP}}$ remain the same. However, we do need one additional typing rule, **[T-Sym]** in fig. 6.7, to type symbols. The type of symbols is kept track of in the environment Γ.

6.4.2 Inputs

In symbolic execution, we do not know what the input of a program will be. In our case this means that we do not know which events will be sent to editors. This is reflected in the definition of symbolic inputs and actions in fig. 6.8. Inputs are still the same and consist of paths and actions. Paths are tagged with one or more F (first) and S (second) tags. Actions no longer contain concrete values, but only symbols. This means that instead of concrete values, editors can only hold symbols.

6.4.3 Path Constraints

Concrete execution of $\widehat{\text{TOP}}$ programs is driven by concrete inputs, which select one branch of conditionals, or make a choice. Since no concrete information is available during symbolic execution, the symbolic execution semantics records how each execution path depends on the symbolic input. This is done by means of path constraints. They can contain symbols, constants, pairs, lists, and operations on them. Figure 6.9 lists the syntax of path constraints.

[E-Edit]

$$\frac{e, \sigma \downarrow v, \sigma'}{\square\, e, \sigma \downarrow \square\, v, \sigma'}$$

[SE-Edit]

$$\frac{\tilde{e}, \tilde{\sigma} \wr \overline{\tilde{v}, \tilde{\sigma}', \varphi}}{\square\, \tilde{e}, \tilde{\sigma} \wr \overline{\square\, \tilde{v}, \tilde{\sigma}', \varphi}}$$

Figure 6.10: Evaluating editors, concrete (left) and symbolic (right)

6.5 Semantics

In this section we discuss the symbolic execution semantics for $\widehat{\text{TOP}}$. The structure of the symbolic semantics closely resembles that of the concrete semantics. It consists of three layers, a big-step symbolic evaluation semantics for the host language, a big-step symbolic normalisation semantics for the task language, and a small-step interaction semantics that processes user inputs. They are described in the following sections. We will study their interesting aspects, and the changes made with respect to the concrete semantics.

6.5.1 Symbolic Evaluation

The host language is a simply-typed lambda calculus with references and basic operations. Most of the symbolic evaluation rules closely resemble the concrete semantics. The original evaluation relation ($\downarrow$) had the form $e, \sigma \downarrow v, \sigma'$, where an expression e in a state σ evaluates to a value v in state σ'. The new relation ($\wr$) adds path constraints φ to the output and has the form $\tilde{e}, \tilde{\sigma} \wr \overline{\tilde{v}, \tilde{\sigma}', \varphi}$. The tilde distinguishes the symbolic variants from the concrete ones. The symbolic semantics can generate multiple outcomes. This is denoted in the evaluation with a line over the result, which can be read as:

$$\overline{\tilde{v}, \tilde{\sigma}', \varphi} = \{(\tilde{v}_1, \tilde{\sigma}'_1, \varphi_1), \cdots, (\tilde{v}_n, \tilde{\sigma}'_n, \varphi_n)\}.$$

The set that results from symbolic execution can be interpreted as follows. Each element is a possible endpoint in the execution of a task. It is guarded by a constraint φ over the symbolic input. Execution only arrives at the symbolic value $\tilde{v}$ and symbolic state $\tilde{\sigma}'$ when the path constraint φ is satisfied.

To illustrate the difference between concrete and symbolic evaluation, fig. 6.10 lists one rule from the concrete semantics and its corresponding symbolic counterpart. The **[E-Edit]** rule evaluates the expression held in an editor to a value. The **[SE-Edit]** does the same, but since it is concerned with symbolic execution, the expression can contain symbols. We therefore do not know beforehand which concrete value will be produced, or even which path the execution will take. If the expression contains a conditional that depends on a symbol, there can be multiple possible result values.

The full symbolic evaluation semantics is listed in fig. 6.11. Most symbolic rules closely resemble their concrete counterparts, and follow directly from them. The most interesting rule is the one for conditionals. The concrete semantics has two separate rules for the then- and the else-branch. The symbolic semantics has one combined rule **[SE-If]**. Since $\tilde{e}_1$ can contain symbols, it can evaluate to multiple values. The rule keeps track of all options.

$$\boxed{\tilde{e}, \tilde{\sigma} \wr \overline{\tilde{v}, \tilde{\sigma}', \varphi}}$$

[SE-Value]

$$\frac{}{\tilde{v}, \tilde{\sigma} \wr \tilde{v}, \tilde{\sigma}, \mathsf{True}}$$

[SE-First]

$$\frac{\tilde{e}, \tilde{\sigma} \wr \overline{\langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}', \varphi}}{\mathsf{fst}\, \tilde{e}, \tilde{\sigma} \wr \overline{\tilde{v}_1, \tilde{\sigma}', \varphi}}$$

[SE-Second]

$$\frac{\tilde{e}, \tilde{\sigma} \wr \overline{\langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}', \varphi}}{\mathsf{snd}\, \tilde{e}, \tilde{\sigma} \wr \overline{\tilde{v}_2, \tilde{\sigma}', \varphi}}$$

[SE-Pair]

$$\frac{\tilde{e}_1, \tilde{\sigma} \wr \overline{\tilde{v}_1, \tilde{\sigma}', \varphi_1} \quad \tilde{e}_2, \tilde{\sigma}' \wr \overline{\tilde{v}_2, \tilde{\sigma}'', \varphi_2}}{\langle \tilde{e}_1, \tilde{e}_2 \rangle, \tilde{\sigma} \wr \overline{\langle \tilde{v}_1, \tilde{v}_2 \rangle, \tilde{\sigma}'', \varphi_1 \wedge \varphi_2}}$$

[SE-Cons]

$$\frac{\tilde{e}_1, \tilde{\sigma} \wr \overline{\tilde{v}_1, \tilde{\sigma}', \varphi_1} \quad \tilde{e}_2, \tilde{\sigma}' \wr \overline{\tilde{v}_2, \tilde{\sigma}'', \varphi_2}}{\tilde{e}_1 :: \tilde{e}_2, \tilde{\sigma} \wr \overline{\tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}'', \varphi_1 \wedge \varphi_2}}$$

[SE-Head]

$$\frac{\tilde{e}, \tilde{\sigma} \wr \overline{\tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}', \varphi}}{\mathsf{head}\, \tilde{e}, \tilde{\sigma} \wr \overline{\tilde{v}_1, \tilde{\sigma}', \varphi}}$$

[SE-Tail]

$$\frac{\tilde{e}, \tilde{\sigma} \wr \overline{\tilde{v}_1 :: \tilde{v}_2, \tilde{\sigma}', \varphi}}{\mathsf{tail}\, \tilde{e}, \tilde{\sigma} \wr \overline{\tilde{v}_2, \tilde{\sigma}', \varphi}}$$

[SE-Deref]

$$\frac{\tilde{e}, \tilde{\sigma} \wr \overline{l, \tilde{\sigma}', \varphi}}{!\tilde{e}, \tilde{\sigma} \wr \overline{\tilde{\sigma}'(l), \tilde{\sigma}', \varphi}}$$

[SE-App]

$$\frac{\tilde{e}_1, \tilde{\sigma} \wr \overline{\lambda x : \tau.\tilde{e}_1', \tilde{\sigma}', \varphi_1} \quad \tilde{e}_2, \tilde{\sigma}' \wr \overline{\tilde{v}_2, \tilde{\sigma}'', \varphi_2} \quad \tilde{e}_1'[x \mapsto \tilde{v}_2], \tilde{\sigma}'' \wr \overline{\tilde{v}_1, \tilde{\sigma}''', \varphi_3}}{\tilde{e}_1 \tilde{e}_2, \tilde{\sigma} \wr \overline{\tilde{v}_1, \tilde{\sigma}''', \varphi_1 \wedge \varphi_2 \wedge \varphi_3}}$$

[SE-If]

$$\frac{\tilde{e}_1, \tilde{\sigma} \wr \overline{\tilde{v}_1, \tilde{\sigma}', \varphi_1} \quad \tilde{e}_2, \tilde{\sigma}' \wr \overline{\tilde{v}_2, \tilde{\sigma}'', \varphi_2} \quad \tilde{e}_3, \tilde{\sigma}' \wr \overline{\tilde{v}_3, \tilde{\sigma}''', \varphi_3}}{\textbf{if}\ \tilde{e}_1\ \textbf{then}\ \tilde{e}_2\ \textbf{else}\ \tilde{e}_3, \tilde{\sigma} \wr \overline{\tilde{v}_2, \tilde{\sigma}'', \varphi_1 \wedge \varphi_2 \wedge \tilde{v}_1} \cup \overline{\tilde{v}_3, \tilde{\sigma}''', \varphi_1 \wedge \varphi_3 \wedge \neg \tilde{v}_1}}$$

[SE-Edit]

$$\frac{\tilde{e}, \tilde{\sigma} \wr \overline{\tilde{v}, \tilde{\sigma}', \varphi}}{\square \tilde{e}, \tilde{\sigma} \wr \overline{\square \tilde{v}, \tilde{\sigma}', \varphi}}$$

[SE-Assign]

$$\frac{\tilde{e}_1, \tilde{\sigma} \wr \overline{l, \tilde{\sigma}', \varphi_1} \quad \tilde{e}_2, \tilde{\sigma}' \wr \overline{\tilde{v}_2, \tilde{\sigma}'', \varphi_2}}{\tilde{e}_1 := \tilde{e}_2, \tilde{\sigma} \wr \overline{\langle\rangle, \tilde{\sigma}''[l \mapsto \tilde{v}_2], \varphi_1 \wedge \varphi_2}}$$

[SE-Fail]

$$\frac{}{\lightning, \tilde{\sigma} \wr \lightning, \tilde{\sigma}, \mathsf{True}}$$

[SE-Then]

$$\frac{\tilde{e}_1, \tilde{\sigma} \wr \overline{\tilde{t}_1, \tilde{\sigma}', \varphi}}{\tilde{e}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \wr \overline{\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \varphi}}$$

[SE-Next]

$$\frac{\tilde{e}_1, \tilde{\sigma} \wr \overline{\tilde{t}_1, \tilde{\sigma}', \varphi}}{\tilde{e}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \wr \overline{\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}', \varphi}}$$

[SE-And]

$$\frac{\tilde{e}_1, \tilde{\sigma} \wr \overline{\tilde{t}_1, \tilde{\sigma}', \varphi_1} \quad \tilde{e}_2, \tilde{\sigma}' \wr \overline{\tilde{t}_2, \tilde{\sigma}'', \varphi_2}}{\tilde{e}_1 \bowtie \tilde{e}_2, \tilde{\sigma} \wr \overline{\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma}'', \varphi_1 \wedge \varphi_2}}$$

[SE-Ref]

$$\frac{\tilde{e}, \tilde{\sigma} \wr \overline{\tilde{v}, \tilde{\sigma}', \varphi} \quad l \notin Dom(\sigma')}{\textbf{ref}\ \tilde{e}, \tilde{\sigma} \wr \overline{l, \tilde{\sigma}'[l \mapsto \tilde{v}], \varphi}}$$

[SE-Or]

$$\frac{\tilde{e}_1, \tilde{\sigma} \wr \overline{\tilde{t}_1, \tilde{\sigma}', \varphi_1} \quad \tilde{e}_2, \tilde{\sigma}' \wr \overline{\tilde{t}_2, \tilde{\sigma}'', \varphi_2}}{\tilde{e}_1 \blacklozenge \tilde{e}_2, \tilde{\sigma} \wr \overline{\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}'', \varphi_1 \wedge \varphi_2}}$$

[SE-Update]

$$\frac{\tilde{e}, \tilde{\sigma} \wr \overline{l, \tilde{\sigma}', \varphi}}{\blacksquare \tilde{e}, \tilde{\sigma} \wr \overline{\blacksquare l, \tilde{\sigma}', \varphi}}$$

Figure 6.11: The evaluation semantics of Symbolic $\widehat{\text{TOP}}$

$$
\begin{aligned}
&\mathscr{F} : \text{Task} \times \text{State} \to \text{Bool} \\
&\mathscr{F}\,(\square\,\tilde{v}, \tilde{\sigma}) &&= \mathsf{False} \\
&\mathscr{F}\,(\boxtimes\,\beta, \tilde{\sigma}) &&= \mathsf{False} \\
&\mathscr{F}\,(\blacksquare\,l, \tilde{\sigma}) &&= \mathsf{False} \\
&\mathscr{F}\,(\lightning, \tilde{\sigma}) &&= \mathsf{True} \\
&\mathscr{F}\,(\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma}) &&= \mathscr{F}\,(\tilde{t}_1, \tilde{\sigma}) \\
&\mathscr{F}\,(\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma}) &&= \mathscr{F}\,(\tilde{t}_1, \tilde{\sigma}) \\
&\mathscr{F}\,(\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma}) &&= \mathscr{F}\,(\tilde{t}_1, \tilde{\sigma}) \wedge \mathscr{F}\,(\tilde{t}_2, \tilde{\sigma}) \\
&\mathscr{F}\,(\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma}) &&= \mathscr{F}\,(\tilde{t}_1, \tilde{\sigma}) \wedge \mathscr{F}\,(\tilde{t}_2, \tilde{\sigma}) \\
&\mathscr{F}\,(\tilde{e}_1 \Diamond \tilde{e}_2, \tilde{\sigma}) &&= \bigwedge \left(\{\, \mathscr{F}\,(\tilde{t}_1, \tilde{\sigma}'_1) \mid \tilde{e}_1, \tilde{\sigma} \twoheadrightarrow \overline{\tilde{t}_1, \tilde{\sigma}'_1} \,\} \cup \{\, \mathscr{F}\,(\tilde{t}_2, \tilde{\sigma}'_2) \mid e_2, \tilde{\sigma} \twoheadrightarrow \overline{\tilde{t}_2, \tilde{\sigma}'_2} \,\} \right)
\end{aligned}
$$

Figure 6.12: Task failing observation function $\mathscr{F}$

It calculates the then-branch, and records in the path constraint that execution can only reach this branch if $\tilde{v}_1$ becomes True. The rule does the same for the else-branch, except it requires that $\tilde{v}_1$ becomes False. Note that both $\tilde{e}_2$ and $\tilde{e}_3$ are evaluated using the same state $\tilde{\sigma}'$, which is the resulting state after evaluating $\tilde{e}_1$.

6.5.2 Observations

The symbolic normalisation and interaction semantics make use of observations on tasks, just like their concrete counterparts. The partial function $\mathscr{V}$ can be used to observe the value of a task. Its definition is unchanged with respect to the original. The function $\mathscr{F}$ observes if a task is failing. Its definition is given in fig. 6.12. A task is failing if it is the fail task ($\lightning$), or if it consists of only failing tasks. This function differs from its concrete counterpart in the clause for user choice. As symbolic normalisation can yield multiple results, all of the results must be failing to make a user choice failing.

6.5.3 Normalisation and Striding

Normalization ($\twoheadrightarrow$) reduces tasks until they are ready to receive input. Very little has to be changed to accommodate symbolic execution. Just like the evaluation semantics, it now gathers sets of results, each result guarded by a path constraint. Figure 6.13 lists the normalisation semantics.

Normalisation makes use of the small-step striding semantics ($\tilde{\mapsto}$). Figure 6.14 lists the symbolic striding semantics. Striding has axioms (**[SS-Edit]**, **[SS-Fill]**, and **[SS-Update]**), which is uncommon for a small-step semantics. Usually, small-step semantics come with an iterated variant that continues stepping until no more rules apply. Our case is different however, because our iterated variant ($\twoheadrightarrow$) alternates between evaluation ($\wr$) and striding ($\tilde{\mapsto}$), and it should be possible that either is the last step. The stopping criterion $\tilde{\sigma}' = \tilde{\sigma}'' \wedge \tilde{t} = \tilde{t}'$, together with the axioms of $\tilde{\mapsto}$ make this possible.

6.5.4 Handling

The handling semantics ($\rightsquigarrow$) deals with user input. In the symbolic case there are symbols instead of concrete inputs. A complete overview of the rules can be found in fig. 6.15.

$$\boxed{\tilde{e}, \tilde{\sigma} \lll \overline{\tilde{t}, \tilde{\sigma}', \varphi}}$$

[SN-Done]

$$\frac{\tilde{e}, \tilde{\sigma} \ll \overline{\tilde{t}, \tilde{\sigma}', \varphi_1} \qquad \tilde{t}, \tilde{\sigma}' \tilde{\mapsto} \overline{\tilde{t}', \tilde{\sigma}'', \varphi_2} \qquad \tilde{\sigma}' = \tilde{\sigma}'' \wedge \tilde{t} = \tilde{t}'}{\tilde{e}, \tilde{\sigma} \lll \overline{\tilde{t}, \tilde{\sigma}', \varphi_1 \wedge \varphi_2}}$$

[SN-Repeat]

$$\frac{\tilde{e}, \tilde{\sigma} \ll \overline{\tilde{t}, \tilde{\sigma}', \varphi_1} \qquad \tilde{t}, \tilde{\sigma}' \tilde{\mapsto} \overline{\tilde{t}', \tilde{\sigma}'', \varphi_2} \qquad \tilde{\sigma}' \neq \tilde{\sigma}'' \vee \tilde{t} \neq \tilde{t}' \qquad \tilde{t}', \tilde{\sigma}'' \lll \overline{\tilde{t}'', \tilde{\sigma}''', \varphi_3}}{\tilde{e}, \tilde{\sigma} \lll \overline{\tilde{t}'', \tilde{\sigma}''', \varphi_1 \wedge \varphi_2 \wedge \varphi_3}}$$

Figure 6.13: Symbolic normalisation semantics

The three rules for the editors (**[SH-Change]**, **[SH-Fill]**, **[SH-Update]**) show how symbols enter the symbolic execution. For example, **[SH-Change]** generates a fresh symbol s and returns an editor containing it. All three rules must perform some type checking at runtime, because they must verify that the given input symbol has the correct type. **[SH-Update]** in particular needs type information about all previously entered symbols, because it checks $\tilde{\sigma}(l) : \beta$, and there may be earlier symbolic inputs in the symbolic value $\tilde{\sigma}(l)$. This bookkeeping is not shown here.

There are several task combinators whose result depends on user input. For example, the parallel combinator ($\bowtie$) receives an input for either the left or the right branch. To accommodate for all possibilities, the **[SH-And]** rule generates both cases. It tags the inputs for the first branch with F and inputs for the second branch with S.

The same principle applies to the external choice combinator ($\Diamond$). The three rules **[SH-PickLeft]**, **[SH-PickRight]**, and **[SH-Pick]** are needed to disallow choosing failing tasks. There is one rule for the case where only the right is failing, one rule when the left is failing, and one for when none of the options are failing.

After input has been handled, tasks are normalised. The combination of those two steps is taken care of by the interact ($\approx\!\!\!\!\!\rightarrow$) semantics, listed in fig. 6.17.

6.5.5 Simulating

The symbolic interaction semantics ($\approx\!\!\!\!\!\rightarrow$) is a small-step semantics. Every step simulates a single symbolic input. To compute every possible execution, including lists of possible inputs, the interact semantics needs to be applied repeatedly, until the task is done. We define a task to be done when it has an observable value: $\mathcal{V}(t', \sigma') \neq \bot$. The simulation function listed in fig. 6.16 is recursively called to produce a list of end states and path constraints. It accumulates all symbolic inputs and returns for each possible execution the observable task value v, the path constraint φ, and the state σ. We consider a task, state and path constraint to be an end state if the task value can be observed, and the path constraint is satisfiable. We write $\mathcal{S}(\varphi)$ if φ is satisfiable. The recursion terminates when one of the following conditions is met.

$$\boxed{\tilde{t}, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}', \tilde{\sigma}', \varphi}}$$

[SS-Edit]

$$\overline{\square\, \tilde{v}, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \square\, \tilde{v}, \tilde{\sigma}, \mathsf{True}}$$

[SS-Fill]

$$\overline{\boxtimes\, \beta, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \boxtimes\, \beta, \tilde{\sigma}, \mathsf{True}}$$

[SS-Update]

$$\overline{\blacksquare\, l, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \blacksquare\, l, \tilde{\sigma}, \mathsf{True}}$$

[SS-ThenCont]

$$\frac{\tilde{t}_1, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1', \tilde{\sigma}', \varphi_1} \qquad \mathscr{V}(\tilde{t}_1', \tilde{\sigma}') = \tilde{v}_1 \qquad \tilde{e}_2 \tilde{v}_1, \tilde{\sigma}' \wr \overline{\tilde{t}_2, \tilde{\sigma}'', \varphi_2} \qquad \neg \mathscr{F}(\tilde{t}_2, \tilde{\sigma}'')}{\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{t_2, \sigma'', \varphi_1 \wedge \varphi_2}}$$

[SS-ThenFail]

$$\frac{\tilde{t}_1, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1', \tilde{\sigma}', \varphi} \qquad \mathscr{V}(\tilde{t}_1', \tilde{\sigma}') = \tilde{v}_1 \qquad \tilde{e}_2 \tilde{v}_1, \tilde{\sigma}' \wr \overline{\tilde{t}_2, \tilde{\sigma}'', \_} \qquad \mathscr{F}(\tilde{t}_2, \tilde{\sigma}'')}{\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1' \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \varphi}}$$

[SS-ThenStay]

$$\frac{\tilde{t}_1, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1', \tilde{\sigma}', \varphi} \qquad \mathscr{V}(\tilde{t}_1', \tilde{\sigma}') = \bot}{\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1' \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \varphi}}$$

[SS-OrNone]

$$\frac{\tilde{t}_1, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1', \tilde{\sigma}', \varphi_1} \qquad \mathscr{V}(\tilde{t}_1', \tilde{\sigma}') = \bot \qquad \tilde{t}_2, \tilde{\sigma}' \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_2', \tilde{\sigma}'', \varphi_2} \qquad \mathscr{V}(\tilde{t}_2', \tilde{\sigma}'') = \bot}{\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1' \blacklozenge \tilde{t}_2', \tilde{\sigma}'', \varphi_1 \wedge \varphi_2}}$$

[SS-OrLeft]

$$\frac{\tilde{t}_1, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1', \tilde{\sigma}', \varphi} \qquad \mathscr{V}(\tilde{t}_1', \tilde{\sigma}') = \tilde{v}_1}{\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1', \tilde{\sigma}', \varphi}}$$

[SS-OrRight]

$$\frac{\tilde{t}_1, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1', \tilde{\sigma}', \varphi_1} \qquad \mathscr{V}(\tilde{t}_1', \tilde{\sigma}') = \bot \qquad \tilde{t}_2, \tilde{\sigma}' \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_2', \tilde{\sigma}'', \varphi_2} \qquad \mathscr{V}(\tilde{t}_2', \tilde{\sigma}'') = \tilde{v}_2}{\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_2', \tilde{\sigma}'', \varphi_1 \wedge \varphi_2}}$$

[SS-Fail]

$$\overline{\lightning, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \lightning, \tilde{\sigma}, \mathsf{True}}$$

[SS-Xor]

$$\overline{\tilde{e}_1 \lozenge \tilde{e}_2, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \tilde{e}_1 \lozenge \tilde{e}_2, \tilde{\sigma}, \mathsf{True}}$$

[SS-Next]

$$\frac{\tilde{t}_1, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1', \tilde{\sigma}', \varphi}}{\tilde{t}_1 \triangleright \tilde{e}_2, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1' \triangleright \tilde{e}_2, \tilde{\sigma}', \varphi}}$$

[SS-And]

$$\frac{\tilde{t}_1, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1', \tilde{\sigma}', \varphi_1} \qquad \tilde{t}_2, \tilde{\sigma}' \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_2', \tilde{\sigma}'', \varphi_2}}{\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma} \mathrel{\tilde{\mapsto}} \overline{\tilde{t}_1' \bowtie \tilde{t}_2', \tilde{\sigma}'', \varphi_1 \wedge \varphi_2}}$$

Figure 6.14: The striding semantics of Symbolic $\widehat{\text{TOP}}$

$$\boxed{\tilde{t}, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}', \tilde{\sigma}', \tilde{\iota}, \varphi}}$$

[SH-Change]

$$\frac{\text{fresh } s \qquad \tilde{v}, s : \beta}{\square\, \tilde{v}, \tilde{\sigma} \rightsquigarrow \square\, s, \tilde{\sigma}, s, \mathsf{True}}$$

[SH-Update]

$$\frac{\text{fresh } s \qquad \tilde{\sigma}(l), s : \beta}{\blacksquare\, l, \tilde{\sigma} \rightsquigarrow \blacksquare\, l, \tilde{\sigma}[l \mapsto s], s, \mathsf{True}}$$

[SH-Fill]

$$\frac{\text{fresh } s \qquad s : \beta}{\boxtimes\, \beta, \tilde{\sigma} \rightsquigarrow \square\, s, \tilde{\sigma}, s, \mathsf{True}}$$

[SH-PassNext]

$$\frac{\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_1', \tilde{\sigma}', \tilde{\iota}, \varphi} \qquad \mathscr{V}(\tilde{t}_1', \tilde{\sigma}') = \bot}{\tilde{t}_1 \rhd \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_1' \rhd \tilde{e}_2, \tilde{\sigma}', \tilde{\iota}, \varphi}}$$

[SH-PassThen]

$$\frac{\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_1', \tilde{\sigma}', \tilde{\iota}, \varphi}}{\tilde{t}_1 \blacktriangleright \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_1' \blacktriangleright \tilde{e}_2, \tilde{\sigma}', \tilde{\iota}, \varphi}}$$

[SH-PassNextFail]

$$\frac{\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_1', \tilde{\sigma}_1, \tilde{\iota}, \varphi} \qquad \mathscr{V}(\tilde{t}_1', \tilde{\sigma}_1) = \tilde{v}_1 \qquad \tilde{e}_2 \tilde{v}_1, \tilde{\sigma}_1 \lhd\!\rhd \overline{\tilde{t}_2, \tilde{\sigma}_2, \_} \qquad \mathscr{F}(\tilde{t}_2, \tilde{\sigma}_2)}{\tilde{t}_1 \rhd \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_1' \rhd \tilde{e}_2, \tilde{\sigma}_1, \tilde{\iota}, \varphi}}$$

[SH-And]

$$\frac{\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_1', \tilde{\sigma}_1, \tilde{\iota}_1, \varphi_1} \qquad \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_2', \tilde{\sigma}_2, \tilde{\iota}_2, \varphi_2}}{\tilde{t}_1 \bowtie \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \begin{array}{r} \overline{\tilde{t}_1' \bowtie \tilde{t}_2, \tilde{\sigma}_1, \mathsf{F}\, \tilde{\iota}_1, \varphi_1} \\ \cup\, \overline{\tilde{t}_1 \bowtie \tilde{t}_2', \tilde{\sigma}_2, \mathsf{S}\, \tilde{\iota}_2, \varphi_2} \end{array}}$$

[SH-Next]

$$\frac{\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_1', \tilde{\sigma}_1, \tilde{\iota}, \varphi_1} \qquad \mathscr{V}(\tilde{t}_1, \tilde{\sigma}) = \tilde{v}_1 \qquad \tilde{e}_2 \tilde{v}_1, \tilde{\sigma} \lhd\!\rhd \overline{\tilde{t}_2, \tilde{\sigma}_2, \varphi_2} \qquad \neg \mathscr{F}(\tilde{t}_2, \tilde{\sigma}_2)}{\tilde{t}_1 \rhd \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \begin{array}{r} \overline{\tilde{t}_1' \rhd \tilde{e}_2, \tilde{\sigma}_1, \tilde{\iota}, \varphi_1} \\ \cup\, \overline{\tilde{t}_2, \tilde{\sigma}_2, \mathsf{C}, \varphi_2} \end{array}}$$

[SH-Or]

$$\frac{\tilde{t}_1, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_1', \tilde{\sigma}_1, \tilde{\iota}_1, \varphi_1} \qquad \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_2', \tilde{\sigma}_2, \tilde{\iota}_2, \varphi_2}}{\tilde{t}_1 \blacklozenge \tilde{t}_2, \tilde{\sigma} \rightsquigarrow \begin{array}{r} \overline{\tilde{t}_1' \blacklozenge \tilde{t}_2, \tilde{\sigma}_1, \mathsf{F}\, \tilde{\iota}_1, \varphi_1} \\ \cup\, \overline{\tilde{t}_1 \blacklozenge \tilde{t}_2', \tilde{\sigma}_2, \mathsf{S}\, \tilde{\iota}_2, \varphi_2} \end{array}}$$

[SH-Pick]

$$\frac{\tilde{e}_1, \tilde{\sigma} \lhd\!\rhd \overline{\tilde{t}_1, \tilde{\sigma}_1, \varphi_1} \qquad \neg \mathscr{F}(\tilde{t}_1, \tilde{\sigma}_1) \qquad \tilde{e}_2, \tilde{\sigma} \lhd\!\rhd \overline{\tilde{t}_2, \tilde{\sigma}_2, \varphi_2} \qquad \neg \mathscr{F}(\tilde{t}_2, \tilde{\sigma}_2)}{\tilde{e}_1 \lozenge \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_1, \tilde{\sigma}_1, \mathsf{L}, \varphi_1} \cup \overline{\tilde{t}_2, \tilde{\sigma}_2, \mathsf{R}, \varphi_2}}$$

[SH-PickLeft]

$$\frac{\tilde{e}_1, \tilde{\sigma} \lhd\!\rhd \overline{\tilde{t}_1, \tilde{\sigma}_1, \varphi_1} \qquad \neg \mathscr{F}(\tilde{t}_1, \tilde{\sigma}_1) \qquad \tilde{e}_2, \tilde{\sigma} \lhd\!\rhd \overline{\tilde{t}_2, \tilde{\sigma}_2, \varphi_2} \qquad \mathscr{F}(\tilde{t}_2, \tilde{\sigma}_2)}{\tilde{e}_1 \lozenge \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_1, \tilde{\sigma}_1, \mathsf{L}, \varphi_1}}$$

[SH-PickRight]

$$\frac{\tilde{e}_1, \tilde{\sigma} \lhd\!\rhd \overline{\tilde{t}_1, \tilde{\sigma}_1, \varphi_1} \qquad \mathscr{F}(\tilde{t}_1, \tilde{\sigma}_1) \qquad \tilde{e}_2, \tilde{\sigma} \lhd\!\rhd \overline{\tilde{t}_2, \tilde{\sigma}_2, \varphi_2} \qquad \neg \mathscr{F}(\tilde{t}_2, \tilde{\sigma}_2)}{\tilde{e}_1 \lozenge \tilde{e}_2, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}_2, \tilde{\sigma}_2, \mathsf{R}, \varphi_2}}$$

Figure 6.15: Symbolic handling semantics

$$
\begin{aligned}
&\mathit{simulate} : \text{Task} \times \text{State} \times [\text{Input}] \times \text{Predicate} \\
&\qquad\qquad \to \mathcal{P}(\text{Value} \times [\text{Input}] \times \text{Predicate}) \\
&\mathit{simulate}(t, \sigma, I, \varphi) = \\
&\bigcup \{\, \mathit{simulate}'(\mathsf{True}, t, t', \sigma', I +\!\!+ [i'], \varphi \wedge \varphi') \mid t, \sigma \approx\!\!\Rightarrow t', \sigma', i', \varphi' \,\} \\
&\mathit{simulate}' : \text{Bool} \times \text{Task} \times \text{Task} \times \text{State} \times [\text{Input}] \times \text{Predicate} \\
&\qquad\qquad \to \mathcal{P}(\text{Value} \times [\text{Input}] \times \text{Predicate}) \\
&\mathit{simulate}'(\mathit{again}, t, t', \sigma', I, \varphi) \\
&\quad \mid \neg \mathcal{S}(\varphi) && \mapsto \emptyset \\
&\quad \mid \mathcal{S}(\varphi) \quad \wedge \mathcal{V}(t', \sigma') = v && \mapsto \{\, (v, I, \varphi) \,\} \\
&\quad \mid \mathcal{S}(\varphi) \quad \wedge \mathcal{V}(t', \sigma') = \bot \wedge t' \neq t && \mapsto \mathit{simulate}(t', \sigma', I, \varphi) \\
&\quad \mid \mathcal{S}(\varphi) \quad \wedge \mathcal{V}(t', \sigma') = \bot \wedge t' = t \wedge \mathit{again} \\
&\quad \mapsto \bigcup \{ \mathit{simulate}'(\mathsf{False}, t', t'', \sigma'', I +\!\!+ [i'], \varphi \wedge \varphi') \mid t', \sigma' \approx\!\!\Rightarrow t'', \sigma'', i', \varphi' \} \\
&\quad \mid \mathcal{S}(\varphi) \quad \wedge \mathcal{V}(t', \sigma') = \bot \wedge t' = t \wedge \neg \mathit{again} \mapsto \emptyset
\end{aligned}
$$

Figure 6.16: Simulation function definition

$$\boxed{\tilde{t}, \tilde{\sigma} \approx\!\!\Rightarrow \overline{\tilde{t}', \tilde{\sigma}', \tilde{\imath}, \varphi}}$$

[SI-Handle]

$$\frac{\tilde{t}, \tilde{\sigma} \rightsquigarrow \overline{\tilde{t}', \tilde{\sigma}', \tilde{\imath}, \varphi_1} \qquad \tilde{t}', \tilde{\sigma}' \lightning \overline{\tilde{t}'', \tilde{\sigma}'', \varphi_2}}{\tilde{t}, \tilde{\sigma} \approx\!\!\Rightarrow \overline{\tilde{t}'', \tilde{\sigma}'', \tilde{\imath}, \varphi_1 \wedge \varphi_2}}$$

Figure 6.17: Symbolic interaction semantics

$\neg\mathcal{S}(\varphi)$ When the path constraint cannot be satisfied, we know that all future steps will not be satisfiable either. All future steps will only add more restrictions to the path constraint. No future path constraint will be satisfiable, and we can therefore safely prune the current branch.

$\mathcal{V}(t, \sigma)$ When the current task has a value it is an end state, which we can return.

$\mathcal{V}(t', \sigma') = \bot \wedge t = t' \wedge \neg again$ When the current task does not produce a value, and it is equal to the previous task except from symbol names in editors, the *simulate* function performs one look-ahead step in case the task does proceed when a fresh symbol is entered. This one step look-ahead is encoded by the parameter *again*. When this parameter is set to False, one step look-ahead has been performed and *simulate* does not continue further. If the task has a value it is returned, otherwise the branch is pruned.

To better illustrate how the *simulate* function works, we study how it simulates the program of fig. 6.1. Figure 6.18 gives a schematic overview of the application of *simulate*. First, it calls the interact semantics to calculate what input the task takes. Users can enter a fresh symbol s_0, as listed on the left. The symbolic execution then branches, since it reaches a conditional. Two branches are generated, one for $s_0 > 0$, and one for $s_0 \leq 0$. In the first case, the resulting task has a value, and symbolic execution ends returning this value and the input. In the second case, the resulting task does not have a value, and the new task is different from the previous task. Therefore *simulate* is called again.

A fresh symbol s_1 is generated. Again, s_1 can either be greater than zero, or less or equal. In the first case, the resulting task has a value, and the execution ends. In the second case however, the task does not have a value, and we find that the task has not been altered (apart from the new symbol). This results in a call to *simulate*′ with *again* set to False.

Once more a fresh symbol s_2 is generated, and s_2 can be greater than zero, or less or equal. In the first case, the task has a value and we are done. In the second case, it does not have a value, the task again has not changed, but *again* is False and therefore symbolic execution prunes this branch.

This example demonstrates a couple of things. From manual inspection, it is clear that only the first iteration returns an interesting result. When s_0 is greater than zero, the task results in a value that is greater than zero. When the input is less than or equal to zero, simulation continues with the task unchanged. Why does the simulation still proceed then? Since the editor ⊠ changes to □, the tasks are not the same after the first step. This causes *simulate* to run an extra iteration. It finds that the task still does not have a value, but now the task has changed. Then *simulate* performs one look-ahead step, by setting the *again*-parameter to False. When this look-ahead does not return a value, the branch is pruned.

6.5.6 Solving

To check the satisfiability of path constraints $\mathcal{S}(\varphi)$, as well as the properties stated about a program, we make use of an external SMT solver. In the implementation we use Z3, although any other SMT solver supporting SMT-LIB can be used.

For fig. 6.1, we would like to prove that after any input sequence I, the path constraints φ imply that the value v of the resulting task t' is greater than 0.

$$\varphi \implies v > 0 \qquad \textbf{where } v = \mathcal{V}(t', \sigma')$$

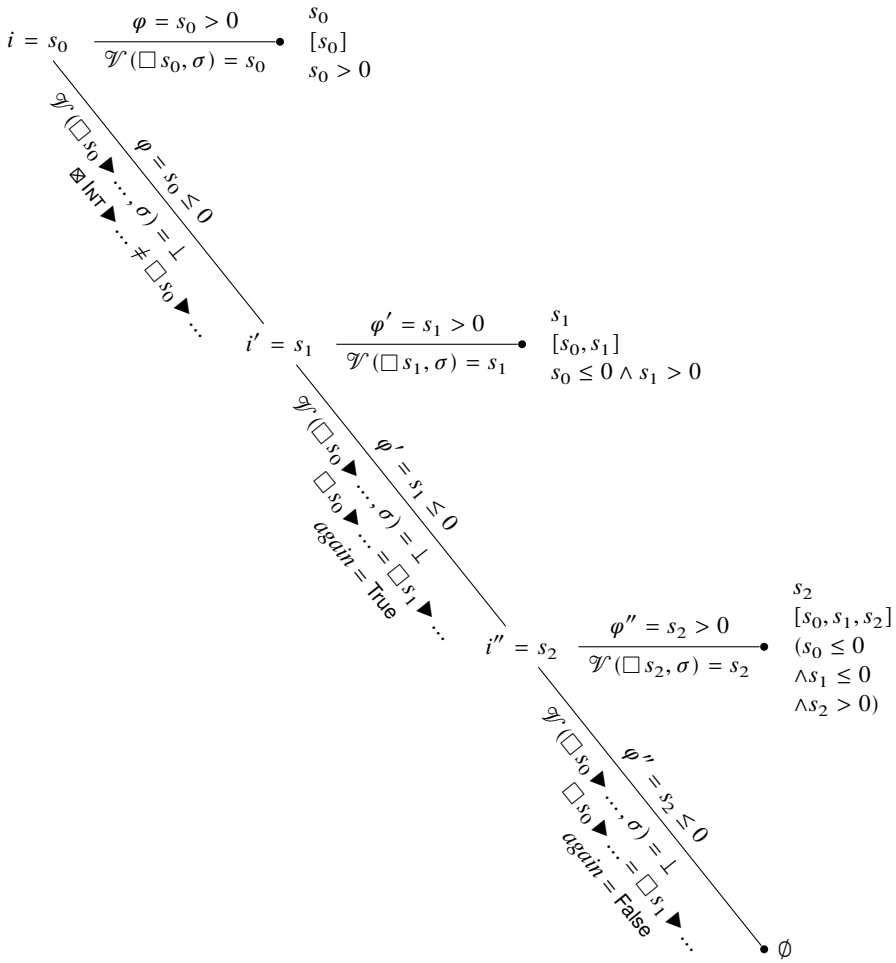


Figure 6.18: Application of the simulation function to fig. 6.1

As shown in fig. 6.18, there are three paths we need to verify. Therefore, we send the following three statements to the SMT solver for verification, which are all easily solvable.

1. $s_0 > 0 \implies s_0 > 0$
2. $s_0 \leq 0 \wedge s_1 > 0 \implies s_1 > 0$
3. $s_0 \leq 0 \wedge s_1 \leq 0 \wedge s_2 > 0 \implies s_2 > 0$

6.5.7 Implementation

We implemented Symbolic $\widehat{\text{TOP}}$ and its symbolic execution semantics in Haskell. With the help of a couple of GHC extensions, the grammar, typing rules and semantics are almost one-to-one translatable into code. Our tool generates execution trees like the one shown in fig. 6.18, which keep track of intermediate normalisations, symbolic inputs, and path constraints. All path constraints are converted to SMT-LIB compatible statements and are verified using the Z3 SMT solver. As of now we do not have a parser, programs must be specified directly as abstract syntax trees.

As is usually the case with symbolic execution, the number of paths grows quickly. The examples in figs. 6.2 and 6.4 generate respectively 2112 and 1166 paths, which takes about a minute to calculate on a contemporary laptop. Solving them is almost instantaneous.

6.5.8 Outlook

Assertions. Other work on symbolic execution often uses assertions, which are included in the program itself. One could imagine an assertion statement **assert** ψ t in $\widehat{\text{TOP}}$ that roughly works as follows. First the SMT solver verifies the property ψ against the current path constraint. If the assertion fails, an error message is generated. Then the program continues with task t. Consider the following small example program.

$$\boxtimes\textsf{Int} \blacktriangleright \lambda x\,.\,\square(\textbf{ref}\ x) \blacktriangleright \lambda l.\ \textbf{assert}\ (!l \equiv x)\ (\square\ \text{"Done"})$$

This program asks the user to enter an integer, which is then stored in a reference. The assertion that follows ensures that the store has been updated correctly.

Assertions have access to all variables in scope, but properties do not. We can give properties access to intermediate variables by returning all values of interest at the end of the program.

$$\boxtimes\textsf{Int} \blacktriangleright \lambda x\,.\,\square(\textbf{ref}\ x) \blacktriangleright \lambda \text{store}\,.\,\square\text{"Done"} \blacktriangleright \lambda \_\,.\,\square(x, !\text{store})$$

It is now possible to verify that the property $\psi\,(x, s) = x \equiv s$ holds. This demonstrates that our approach has expressive power similar to assertions. Having assertions in the language would be more convenient for programmers, and we would like to add them in the future.

Input-dependent predicates. Another feature we would like to support in the future are input-dependent predicates. Consider the following small program.

$$\boxtimes\textsf{Int} \blacktriangleright \lambda x\,.\,\textbf{if}\ x > 0\ \textbf{then}\ \square\text{"Thank you"}\ \textbf{else}\ \square\text{"Error"}$$

The user inputs an integer. If the integer is larger than zero, the program prints a thank you message. If the integer is smaller than zero, an error is returned. If we want to prove that given a positive input, the program never returns "Error", we need to be able to talk about inputs directly in predicates. Currently our symbolic execution does not support this.

6.6 Properties

In this section we describe what it means for the symbolic execution semantics to be correct. We prove it sound and complete with respect to the concrete semantics of $\widehat{\text{TOP}}$. In this thesis we only present the theorems, the full proofs can be found in the PhD thesis of Naus [2020]. To relate the two semantics, we use the concrete inputs listed in chapter 5.

6.6.1 Soundness

To validate the symbolic execution semantics, we want to show that for every individual symbolic execution step there exists a corresponding concrete one. This soundness property is expressed by theorem 6.6.1.

Theorem 6.6.1 (Soundness of interact)**.** *For all concrete tasks t, concrete states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{\iota}', \tilde{\sigma}', \tilde{\iota}, \varphi)$ in $t, \sigma \rightsquigarrow \overline{\tilde{\iota}', \tilde{\sigma}', \tilde{\iota}, \varphi}$ that $M\varphi$ implies $t, \sigma \overset{M\tilde{\iota}}{\Rightarrow} t', \sigma''$ and $M\tilde{\iota}' \equiv t'$ and $M\tilde{\sigma}' \equiv \sigma''$.*

Since the interaction semantics makes use of the handling and the normalisation semantics, we require two lemmas: one showing that the handling semantics is sound, lemma 6.6.2, and one showing that the normalisation semantics is sound, lemma 6.6.3.

Lemma 6.6.2 (Soundness of handling)**.** *For all concrete tasks t, concrete states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{\iota}', \tilde{\sigma}', \tilde{\iota}, \varphi)$ in $t, \sigma \rightsquigarrow \overline{\tilde{\iota}', \tilde{\sigma}', \tilde{\iota}, \varphi}$, that $M\varphi$ implies $t, \sigma \overset{M\tilde{\iota}}{\to} t', \sigma'$ and $M\tilde{\iota}' \equiv t'$ and $M\tilde{\sigma}' \equiv \sigma'$.*

Lemma 6.6.3 (Soundness of normalisation)**.** *For all concrete expressions e, concrete states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{\iota}, \tilde{\sigma}', \varphi)$ in $e, \sigma \lightsquigarrow \overline{\tilde{\iota}, \tilde{\sigma}', \varphi}$, that $M\varphi$ implies $e, \sigma \Downarrow t', \sigma''$ and $M\tilde{\iota} \equiv t'$ and $M\tilde{\sigma}' \equiv \sigma''$.*

Since lemma 6.6.3 makes use of both the striding and the evaluation semantics, we must show soundness for those too.

Lemma 6.6.4 (Soundness of striding)**.** *For all concrete tasks t, concrete states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{\iota}', \tilde{\sigma}', \varphi)$ in $t, \sigma \tilde{\mapsto} \overline{\tilde{\iota}', \tilde{\sigma}', \varphi}$, that $M\varphi$ implies $t, \sigma \mapsto t', \sigma'$ and $M\tilde{\iota}' \equiv t' \wedge M\tilde{\sigma}' \equiv \sigma'$.*

Lemma 6.6.5 (Soundness of evaluation)**.** *For all concrete expressions e, concrete states σ and mappings $M = [s_0 \mapsto c_0, \cdots, s_n \mapsto c_n]$, we have for all tuples $(\tilde{v}, \tilde{\sigma}', \varphi)$ in $e, \sigma \downarrow \overline{\tilde{v}, \tilde{\sigma}', \varphi}$, that $M\varphi$ implies $e, \sigma \downarrow v, \sigma' \wedge M\tilde{v} \equiv v \wedge M\tilde{\sigma}' \equiv \sigma'$.*

6.6.2 Completeness

We want to show that for every concrete execution there exists a symbolic one. To state this Theorem, we require a simulation relation $\tilde{\iota} \sim i$, which means that the symbolic input $\tilde{\iota}$ follows the same direction as the concrete input i. This relation is defined below.

Definition 6.6.6 (Input simulation)**.** A symbolic input $\tilde{\imath}$ simulates a concrete input i denoted as $\tilde{\imath} \sim i$ in the following cases.

$$
\begin{aligned}
s &\sim a && \text{for every symbol } s \text{ and concrete action } a \\
\tilde{\imath} \sim i &\implies \mathsf{F}\,\tilde{\imath} \sim \mathsf{F}\,i \\
\tilde{\imath} \sim i &\implies \mathsf{S}\,\tilde{\imath} \sim \mathsf{S}\,i
\end{aligned}
$$

This allows us to define the completeness property as listed in theorem 6.6.7.

Theorem 6.6.7 (Completeness of interact)**.** *For all concrete tasks t, concrete states σ and concrete inputs i such that $t, \sigma \overset{i}{\Rightarrow} t', \sigma'$ there exists an $\tilde{\imath} \sim i$, $\tilde{t}$, $\tilde{\sigma}$ and φ such that $(\tilde{t}, \tilde{\sigma}, \tilde{\imath}, \varphi)$ in $t, \sigma \rightsquigarrow \overline{\tilde{t}, \tilde{\sigma}, \tilde{\imath}, \varphi}$.*

The proof of theorem 6.6.7 is rather simple. We show that handling is complete (lemma 6.6.8) and that the subsequent normalisation is complete (lemma 6.6.9).

Lemma 6.6.8 (Completeness of handling)**.** *For all concrete tasks t, concrete states σ and concrete inputs i such that $t, \sigma \overset{i}{\to} t', \sigma'$ there exists an $\tilde{\imath} \sim i$, $\tilde{t}$, $\tilde{\sigma}$ and φ such that $(\tilde{t}, \tilde{\sigma}, \tilde{\imath}, \varphi)$ in $t, \sigma \rightsquigarrow \overline{\tilde{t}, \tilde{\sigma}, \tilde{\imath}, \varphi}$.*

Since handling makes use of normalisation and evaluation, we need to prove that they too are complete. These properties are listed in lemmas 6.6.9 and 6.6.10

Lemma 6.6.9 (Completeness of normalisation)**.** *For all concrete expressions e and concrete states σ such that $e, \sigma \Downarrow t, \sigma$ there exists a symbolic execution result $(t, \sigma, \mathsf{True})$ in $e, \sigma \lll \overline{\tilde{t}, \tilde{\sigma}, \varphi}$.*

Lemma 6.6.10 (Completeness of evaluation)**.** *For all concrete expressions e and concrete states σ such that $e, \sigma \downarrow v, \sigma$ there exists a symbolic execution result $(v, \sigma, \mathsf{True})$ in $e, \sigma \wr \overline{\tilde{v}, \tilde{\sigma}, \varphi}$.*

6.7 Related Work

Symbolic execution. Symbolic execution [King, 1975; Boyer et al., 1975] is typically applied to imperative programming languages. For example Bucur et al. [2014] prototype a symbolic execution engine for interpreted imperative languages. Cadar et al. [2008] use it to generate test cases for programs that can be compiled to LLVM bytecode. Jaffar et al. [2012] use it for verifying safety properties of C programs. In recent years it has been used for functional programming languages as well. To name some examples, there is ongoing work [Hallahan et al., 2019] to implement a symbolic execution engine for Haskell. Giantsios et al. [2017] use symbolic execution for a mix of concrete and symbolic testing of programs written in a subset of Core Erlang. Their goal is to find executions that lead to a runtime error, either due to an assertion violation or an unhandled exception. Chang et al. [2018] present a symbolic execution engine for a typed lambda calculus with mutable state where only some language constructs recognise symbolic values. They claim that their approach is easier to implement than full symbolic execution and simplifies the burden on the solver, while still considering all execution paths. The difficulty of symbolic execution for functional

languages lies in symbolic higher-order values, that is functions as arguments to other functions. Hallahan et al. [2017] solve this with a technique called *defunctionalization*, which requires all source code to be present, so that a symbolic function can only be one of the present lambda expressions or function definitions. Giantsios et al. [2017] also require all source code to be present, but they only analyze first-order functions. They can execute higher-order functions, but only with concrete arguments. Our method also requires closed well-typed terms, so we never execute a higher-order function in isolation. Furthermore, we currently do not allow functions and tasks as task values. Together, this means that symbolic values can never be functions.

Contracts. Another method for guaranteeing correctness of programs are *contracts*. Contracts refine static types with additional conditions. They are enforced at runtime. Contracts were first presented by Meyer [1992] for the Eiffel programming language. Findler and Felleisen [2002] applied this technique to functional programming by implementing a contract checker for Scheme. Their contracts are assertions for higher-order programs. Contracts can be used to specify properties more fine-grained than what a static type system is capable of. It is possible, for example, to refine the arguments or return values of functions to numbers in a certain range, to positive numbers or non-empty lists. Nguyen et al. [2017] combine contracts and symbolic execution to provide *soft contract checking*. The two ideas go hand in hand in that contracts aid symbolic execution with a language for specifications and properties for symbolic values, and symbolic execution provides compile-time guarantees and test case generation. They present a prototype implementation to verify Racket programs.

Axiomatic program verification. One of the classical methods of proving partial correctness of programs is Hoare's axiomatic approach [Hoare, 1969], which is based on pre- and postconditions. Nielson and Nielson [1992] have a nice introduction to the topic. The axiomatic approach is usually applied to imperative programs, requires manually stating loop invariants, and manually carrying out proofs. Some work has been done to bring the axiomatic method to functional programming. The current state of SMT solving allows for automated extraction and solving of a large amount of proof obligations. Notable works in this field are for example the Hoare Type Theory by Nanevski et al. [2006], the Hoare and Dijkstra Monads by Nanevski et al. [2008]; Swamy et al. [2013], or the Hoare logic for the state monad by Swierstra [2009]. The difference between the work cited here and our work is that the axiomatic method focuses on stateful computations, while we try to incorporate input as well.

6.8 Conclusion

In this chapter, we have demonstrated how to apply symbolic execution to $\widehat{\text{TOP}}$ to verify individual programs. We have developed both a formal system and an implementation of a symbolic execution semantics. Our approach has been validated by proving the formal system correct, and by running the implementation on example programs. For these two example programs, a subsidy request workflow and a flight booking workflow, we have verified that they adhere to their specifications.

6.8.1 Future Work

There are many ways in which we would like to continue this line of work. First, we believe that more can be done with symbolic execution. Our current approach only allows proving predicates over task results and input values. We cannot, however, prove properties that depend on the order of the inputs. Since the symbolic execution currently returns a list of symbolic inputs, we think this extension is feasible.

Second, our symbolic execution only applies to $\widehat{\text{TOP}}$. We would like to see if we can fit it to iTasks. This poses several challenges. iTasks does not have a formal semantics in the sense that $\widehat{\text{TOP}}$ has. The current implementation in Clean is the closest thing available to a formal specification. There are also a few language features in iTasks that are not covered by $\widehat{\text{TOP}}$, for example recursion.

Third, we would like to apply different kinds of analyses altogether. Can a certain part of the program be reached? Does a certain property hold at every point in the program? Are two programs equal? And what does it mean for two programs to be equal? We think that these properties require a different approach.

Acknowledgements

The authors are grateful to Johan Jeuring, Sjaak Smetsers, and Andreas Vinter-Hviid for fruitful discussions, and Rinus Plasmeijer, Peter Achten and Pieter Koopman for proofreading. This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO), and which is partly funded by the Ministry of Economic Affairs.

Part III

Dynamic Resource Management

7 Dynamic Resource and Task Management

Carrying out maritime missions comprises many phases from preparation to execution. In the long term, we would like to have an integrated toolchain that supports the crew at every phase. In this chapter, we study concepts for resource and task management in the execution phase. When the tasks to be executed have been identified, the question arises who should be assigned to them. This is both a scheduling and an assignment problem. We narrow down what kind of problem we have at hand to get an understanding what a first step towards an integrated command and control system could look like. This also enables us to classify our problem with the existing literature on planning and scheduling. We develop a domain model for tasks and resources, their connection via capabilities, together with assessment functions to compare assignments. We study what kind of information would be needed to give useful scheduling advice.

7.1 Introduction

In previous work [Bolderheij et al., 2018; Kool, 2017] we developed prototype Command and Control applications for reasoning about mission goals and tasks to achieve the goals. These applications dealt with assignments of resources to tasks. We studied examples from the maritime domain, that is navy ships with their tasks and resources. The concepts of tasks and resources were defined with the maritime domain in mind, which led to definitions that were too domain-specific. In this chapter we want to generalize some of these concepts, and then specialize them back to the maritime domain. Our concepts should be general enough to be specialized to a variety of different scenarios, and to different levels of granularity inside one scenario.

The core goal of this chapter is the concept of a system that provides interactive decision support for on-line scheduling for command and control on board of navy ships. With this in mind, we define the concepts of tasks and resources, together with a way for tasks to specify their resource requirements. We describe how resources specify which tasks they can execute. We define a quality metric to determine which resources are most suitable for executing a task. Resources can be subject to degradation, which allows modelling broken machines and tired people. These definitions serve as basis for a literature study to identify techniques that are useful to us. We demonstrate how the system would handle an example scenario.

The long term vision of this work is a tool for command and control that takes workflows from a planning phase as input, and finds suitable assignments of resources to tasks so that they can be performed efficiently. A supervisor monitors execution of tasks and stays in contact with the crew to solve problems when they occur. The supervisor has overview

over the current state of affairs, running tasks, busy resources, and remaining capacities. Crew members carry some kind of smart device with an app that displays all tasks they are currently assigned to, in the form of a checklist. They can communicate with the supervisor by marking tasks as *in progress*, *paused*, *done*, or maybe *impossible to progress*. Tasks that require more complex interactions can have an extended user interface that allows, for example, entering data. The app can notify crew members when they are assigned to high priority tasks.

7.2 A Model For Tasks and Resources

The main goal of this work is the development of a model for tasks and resources that is abstract enough to be applicable to a variety of situations and granularities. In this chapter we talk about three kinds of roles, the modeller, the planner, and the supervisor. The modeller identifies tasks, resources, and capabilities in the domain of discourse and formalizes them in our system. Such a formalization is called a *model*. The planner uses the model to solve the problems of a particular scenario. Planning involves picking the right tasks to achieve a goal and assigning the right resources to do so efficiently. Once planning is done and execution of a plan starts, the supervisor monitors progress of the plan and makes adjustments to it as necessary. These three roles are not necessarily performed by different people, nor do they happen at distinct phases. Modelling and planning can go hand in hand, as can planning and supervision.

7.2.1 Running Example: Making Pizza

We illustrate the design decisions of this chapter using a recurring example. The example is simple, but shows many of the issues we want to be able to handle. The example serves to check that our system is expressive enough to model a common scenario, to discuss granularities where it makes sense to stop modelling, and to get an idea about what conclusions can be drawn from some given information. The example will be developed further in subsequent sections, and discussed in more detail in section 7.5. Imagine a family planning their dinner. The family consists of four people, Mom, Dad, Alice and Bob. They decide that they want to make pizza. For the sake of our discussion, making pizza consists of two tasks, preparing the dough and finishing the pizza. Our family has enough ingredients at home to prepare the dough, but for the toppings someone needs to shop groceries. The family owns a car and a bike, but only Mom and Dad have a driver's license.

7.2.2 Tasks and Resources

At the heart of our system lies the assignment of resources to tasks. A *task* is a unit of work that we want to be able to plan and manage. A *resource* is a scarce, uniquely identifiable or quantifiable object that a task must exclusively claim in order to be executed. The important aspect here is that resources must be able to be claimed exclusively. One goal of our system is reasoning about resource conflicts, and conflicts cannot occur when something can be infinitely shared. For example people, hammers, matchsticks, fuel, and storage space can be resources in our model, but information or time cannot.

Georgievski and Aiello [2015] argue that time is a consumable resource. We argue that time cannot be regarded as a resource at all, because time lacks the exclusivity that is essential for something to be considered a resource. Consider time and fuel. If we have five liters of fuel and two tasks that require four liters each, then these two tasks can neither be executed in sequence nor in parallel. If we have a deadline five minutes from now then we can execute as many four-minute tasks as we want in parallel, provided that all other resource requirements are met. Furthermore, time cannot be saved up. If we do not use some fuel now, we still have it later. Time is gone once it has passed, no matter whether we execute a task in it.

Furthermore, we do not differentiate between *performers*, which are the resources that actually perform tasks, and passive resources like tools and materials.

7.2.3 Capabilities and Assignments

When a modeller wants to specify that a task needs some resource, he usually does not specify a particular individual, but rather describes it in abstract terms. For example, the modeller might specify that making pizza requires a cook, but he would not immediately specify that it specifically requires Bob. Furthermore, Bob might have many more skills besides making pizza. To express this indirect coupling between tasks and resources, we introduce the concept of *capabilities*. Capabilities are descriptions of the roles of the resources that a task requires. In turn, resources specify all the roles they are capable of fulfilling. In case modellers want to specify a particular resource, they can do so by using a capability that only one resource has.

Capabilities also always specify a capacity. This way modellers can specify that a task needs a certain quantity of some resource, for example five liters of fuel. There is a difference between one two-capacity capability and two one-capacity capabilities. For example, if a task needs two mechanics, the modeller has to specify two one-capacity capabilities. If the modeller specifies one mechanic with capacity two, then this requirement can never be met, because people have a capacity of one for their capabilities.

During planning, once the desired tasks have been chosen, a scheduler can consider capabilities and some optimization criterion to find an optimal assignment of resources to the tasks. Resources that are assigned to a task are said to be *claimed* by the task.

Figure 7.1 shows a UML class diagram for tasks, resources, and capabilities, and the relations between them. A task can require many capabilities, a resource can have many capabilities, and a task can claim many resources. A resource can be claimed by many tasks if its capacity permits it.

Another aspect we want to be able to capture is partial dependencies between tasks. Sometimes tasks depend on other tasks, which constrains the order in which they can be executed. Dependencies express that a prerequisite task has to be completed before another one can be started. The possibility to model such dependencies is represented in fig. 7.1 by the relation from Task to Task. One task can depend on many other tasks, and one task can be prerequisite for many other tasks.

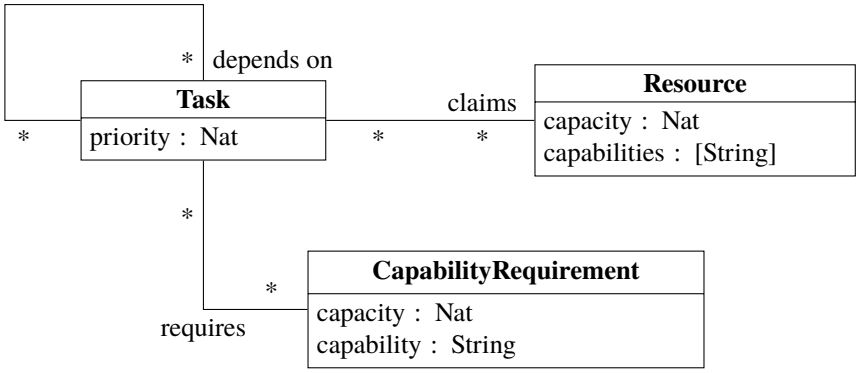


Figure 7.1: UML class diagram for Tasks, Resources, and Capabilities

7.3 Capability Functions

If there are multiple candidate resources that can be assigned to a task, a decision must be made which resource to pick. To do this, we would like to have a quality measure that estimates how good a resource will perform a task. We express quality as a percentage. Quality does not only encode the skill level of a resource, but can be used to encode all kinds of preferences. Our system uses *capability functions* to determine the quality with which a resource can execute a given task. Whether degradation should be factored directly into quality or needs independent handling requires further investigation. The result of a capability function is a percentage. To understand what the input should be, consider the following examples.

Example 7.3.1. On board of a marine ship are two radars. The task is to track a helicopter that has already been detected. Tracking means having high-frequency, high-accuracy updates of the helicopter's position.

Example 7.3.2. A gas pipe is on fire in room A1 on board of a ship. The task is to extinguish the fire. There are two methods available to extinguish the fire: a water mist installation and a dry chemical extinguisher.

Example 7.3.3. For having dinner, someone must shop groceries. On the shopping list is, among other things, some fresh fish. The fishmonger on the weekend market is five kilometers away. There are a bike and a car available.

To determine for each scenario which resources are best suited to execute the task, we must take into account not only the tasks and the resources, but also the helicopter, the fire, and the shopping list. The quality of a radar track depends on the type of the target to be tracked, its distance to the ship, and the weather. The quality of extinguishing a fire depends on the type of the fire and who is closest to the location of the fire. The quality of a person and a vehicle to do shopping depends on the items on the shopping list. In each case, the signature of the capability function must be different, in both the number of arguments and their type. This makes it hard to construct a generic scheduler that, without domain knowledge, can assess the quality of assignments in different scenarios.

To have uniform capability functions, we introduce two new concepts: targets and systems. A *target* is the reason why a task must be executed. The targets in our examples are the contact of type helicopter, with its position and speed, the fire with its location and type, and the shopping list with its contents. Other examples for targets are customer complaints, software bugs, or expected threats.

A *system* is a combination of resources which jointly provide a capability. Systems are needed when the quality of a resource cannot be determined in isolation. Systems have two attributes: the capability they provide and a list of capabilities they require to do so. An assignment of concrete resources to the required capabilities of a system is called a *system assignment*. Single resources can act as systems, in cases where the quality of a single resource can be judged in isolation. In example 7.3.3, it is the combination of driver and vehicle whose suitability for shopping must be judged, not driver and vehicle in isolation. Mom with the car is well suited, while both Bob or Mom with the bike are less suited, and Bob with the car is not suited at all, because Bob doesn't have a driver's license. In example 7.3.2 we have two ways of achieving the same goal. The fire can be extinguished by the water mist installation, or by a person with a fire extinguisher. This can be modelled as a task that requires a single capability *fire brigade*, and two systems that provide this capability. The first system requires a capability *water mist*, while the second system requires two capabilities *fire fighter* and *fire extinguisher*.

Many capability functions also require access to the *environment*, a database with the current operational picture. The environment contains things like the weather conditions and the topology of the surrounding area. All these considerations suggest the following signature for capability functions.

Definition 7.3.4. (Capability function) The quality of a system for a task depends on the system assignment, the task, the capability the system provides for the task, the task's target, and the environment. The expected quality is given as a percentage.

$$\text{capability} : (\text{Assignment}, \text{Task}, \text{Capability}, \text{Target}, \text{Environment}) \to \text{Percentage}$$

7.3.1 Extensibility of the Scheduler

As modellers model scenarios by creating tasks, resources, targets, and systems, they also have to create capability functions. To provide the algorithmic expressiveness some capability functions need, they are best specified in a programming language. This requires modellers to be at least somewhat skilled in programming. There are three common ways to let users specify input with algorithmic content. First we could provide a small domain-specific language, specifically tailored for specifying capability functions. Second, we could make the system extensible by providing an interface to a scripting language like Python or Lua. Third, we could let modellers use the same language the system itself is implemented in. The first option requires substantial implementation effort to provide features of a general-purpose programming language, and even more if modellers want access to external libraries. The second option still requires some implementation effort to make the interface of the scheduler available to the scripting language. For a first prototype implementation, the third option is the best way, as it makes a general-purpose programming language available to modellers at very low implementation overhead, as all interfaces and data structures can readily be accessed.

7.4 Human in the Loop

When a resource becomes degraded during execution of a plan and some tasks can no longer be finished, a change in plan is required. Dealing with unexpected situations often requires flexibility, creativity and improvisation, all abilities where humans perform better than computers. On the other hand, in order to make informed decisions, humans need insight into the current state of affairs. Information needs to be distributed quickly and stored accurately, and big numbers of possibilities must be assessed in a short time, all jobs where computers outperform humans.

We fall into line with Johnson [2014] and Bradshaw et al. [2013] who argue that for effective human-robot collaboration, full autonomy of the team members is not desirable: "Resilience in human-machine systems benefits from a teamwork infrastructure designed to exploit interdependence". They define interdependence as the complementary relations that participants in a joint activity rely on in order to remedy their lacking capabilities. At the core of Johnson's model for human-machine collaboration lies the concept of OPD, which stands for observability, predictability, and directability. Observability means making relevant aspects of one's own status, knowledge and environment available to others. Predictability means that others can rely on their prediction about one's own behavior when planning their actions. Directability means the ability to influence the behavior of others, and being influenced by others.

For our application, observability means that the ship needs an up-to-date picture of the degradation of systems and people on board. The other way around, the ship's internal picture should be accessible to the supervisor. Predictability means that in similar situations, the system should come up with similar plans, and small changes in the environment should lead to small changes in a plan. Directability means that the supervisor can influence planning and always override any decision the system makes. For example the supervisor should be able to make plans that are nonsensical to the system, or start tasks whose resource requirements are not fully met. We propose two categories of actions which allow supervisors to intervene in the execution of a plan. The first category is called *plan B* and includes decisions that can be made before a plan is executed. The second category is called *dynamic planning* and includes actions to modify a plan while it is being executed.

7.4.1 Plan B

A plan B is an alternative plan to achieve the same goal, using different tasks and resources. In the extreme case, plan B uses completely different tasks and resources. Realistically, there is some overlap between plans A and B. If plans A and B overlap at some resource, and this resource becomes degraded, then B also can no longer be executed. Consequently, for an alternative plan to be useful, an estimation of *expected degradation* is required. Plans should not overlap in resources that are likely to degrade. Our system should give insight into resource overlap. The advantage of alternative plans is that they can be activated quickly. Coming up with them however requires additional planning beforehand.

7.4.2 Dynamic Planning

Dynamic planning allows supervisors to change plans while they are being executed. The system should give supervisors the freedom they need to improvise. There are two reasons why a dynamic change in plan becomes necessary. First, a resource required for execution of a running task becomes degraded. Second, a new goal is identified, and new tasks must be executed to achieve it. When a running task becomes disabled because one of its assigned resources degrades, the supervisor has the following options.

- Keep the task running, but change its resource assignment. This is useful for tasks that allow new resources to pick up the work where the degraded resources left it.
- Cancel the current task, start new tasks and assign different resources to them. Cancelling is useful for tasks that cannot be handed over to other resources.
- Pause parts of plan A, start new tasks with new resources to repair the degradation, then continue with plan A.

Both pausing and cancelling a task frees all its assigned resources.

When additional tasks must be started because new goals have been identified, supervisors need similar actions. Goals have priorities, and tasks have priorities derived from them. A *resource conflict* exists if a set of new tasks cannot be executed because there are not enough resources available. Supervisors must find ways to resolve such conflicts, and our system must give them the tools to do so. This includes querying the current status to find solutions, and modifying the current status to implement them. For example, given a new task, the system displays all tasks with lower priority such that when they are paused, their freed resources allow the new task to be executed. The supervisor can then pick some running tasks and either cancel or pause them to free their resources.

7.5 Making Pizza

In this section we look at a situation in the daily life of a four-person family. They are sitting at the breakfast table on a Saturday morning, planning their dinner. We walk through the planning process while identifying relevant tasks and resources. The goal of this study is to make sure that our system is expressive enough to describe the scenario. Furthermore, we discuss the granularities where it makes sense to stop modelling. The family consists of four people, Mom, Dad, Alice and Bob. The family decides that they want to have pizza for dinner today. This suggests that there is a task *make pizza*.

7.5.1 Making Pizza

Whenever we identify a new task, we must consider two questions. First, does it make sense to split the task into subtasks? Second, which resources are required to execute the task? Making pizza consists of many subtasks, as is evident by reading any pizza recipe, and we could go even further and specify subtasks down to the level of individual hand movements. This is of course too fine grained, and even the individual steps in the recipe are too detailed for the purpose of planning dinner. The goal of planning is to provide actors with the necessary instructions to perform their tasks independently.

Being too specific is not useful, and so is being too general. Instead of having a task *make pizza*, it is conceivable to have a generic task *cook meal* that is parameterized by a recipe. The recipe would then determine which subtasks there are and which ingredients and appliances are required to execute the task. At this point, we might as well take the recipe itself as the task, which would make the *cook meal* task useless.

For our scenario we assume that the family members are sufficiently skilled in cooking so that the instruction *make pizza* is enough for them to know what to do. There is however one subtask we do want to make explicit. Preparing the dough is a subtask that can reasonably be performed independently and even by a different person. For the sake of discussion, let's say that there are enough ingredients for the dough, but not for the topping. The family decides that one person can start making the dough, while another person shops for the remaining ingredients. After both these tasks are completed, the pizza can be finished. This suggests three tasks in total: *make dough*, *shop groceries*, and *finish pizza*. The task *finish pizza* depends on both *make dough* and *shop groceries*. These three tasks have the right granularity that allows performers to execute them without further instructions.

7.5.2 Required Resources

The next step in the modelling phase is to identify for each task what resources it needs. The goal is to be reasonably realistic and specify only those resources that are relevant for identifying conflicts. All three tasks need a person to execute them, and we have to specify the required capabilities. There are many possibilities of how precise we want this specification to be. For example, we could be very precise and model that for making dough, we need someone with the capability *make dough*. We could also model that we need a *cook*, or even that we just need a *person*. In this example, we go with the capability *cook* for both tasks *make dough* and *finish pizza*. For the task *shop groceries* we say that we need a *shopper*.

When considering the required materials and tools for the task *make dough*, we again have to think about the right precision. Do we explicitly model every required kitchen appliance, including the oven and individual spoons, or do we just specify that a functional kitchen is needed? Do we specify all the ingredients in the recipe? The answer is again guided by the conflicts we want the system to be able to detect. On the one hand, more information means the system can draw more conclusions. On the other hand, the more precisely we specify the resource requirements of a task, the more work it is to model the resources themselves, and to be useful this information must be kept in sync with the real world. For our scenario, we do not specify the ingredients, but we specify that a functional kitchen is required. We do the same for the task *finish pizza*, and specify that it needs a cook and a kitchen.

At this point, the modeller has to be careful. We just specified that two tasks need the same resource, a kitchen, and if we assume that only one kitchen exists, our system does not allow these tasks to be executed in parallel. In this particular case that is not a problem, because the second task depends on the first one anyway, but there might be situations where we want two tasks in parallel in the same kitchen. Sometimes two people can work in one kitchen on different tasks, so the kitchen is not exclusively claimed by either. In this sense, the kitchen does not qualify as a resource according to our definition. To capture the

Name	Required capabilities
make dough	cook, kitchen
shop groceries	shopper, vehicle
finish pizza	cook, kitchen

Table 7.1: Tasks

Name	Capabilities
Mom	cook, shopper
Dad	cook
Alice	cook, shopper
Bob	shopper
Bike	vehicle
Car	vehicle
Kitchen	kitchen

Table 7.2: Resources

situation in our system, the modeller could split the kitchen into a number of shares, and specify that the cooking subtasks require one share each.

The task *shop groceries*, in addition to a shopper, also requires a vehicle. We assume that the family has two vehicles, a bike and a car. When assigning resources to this task, not all combinations of shoppers and vehicles make sense, because only Mom and Dad have a driver's license. Modellers can exclude impossible combinations of resources with capability functions, which are discussed in section 7.3.

With the information gathered so far, the model is precise enough for the purpose of this example. Tables 7.1 and 7.2 summarize the identified tasks and resources. We could have been more precise, and also modelled that *shop groceries* requires money, or that *finish pizza* requires ingredients for the topping. Again, this comes down to the detail of the instructions the system should give to the people involved.

7.5.3 Scheduling

With the information gathered about the scenario so far, it is possible to construct schedules. A *schedule* consists of two parts. First, a *workflow*, which is a complete specification of the order in which to execute tasks. Second, an *assignment* of resources to the tasks in the workflow. The three tasks of our example and their dependencies permit three possible workflows. They are shown in fig. 7.2. Workflow A specifies that *shop groceries* and *make dough* must be executed in parallel. Parallel execution for us means that they are to be started at the same time and that they cannot share any resources. Workflows B and C specify that all three tasks have to be executed in sequence. In both cases, the same person can be assigned to all of them. With our resources and the requirements for the tasks, there are 42 possible assignments for workflow A, and 54 for each of B and C, which makes a total of 150 ways of making pizza.

Without additional information for assessing schedules, a scheduler can do little more than list all possibilities. A scheduler could optimize for least total resource usage, which means it tries to reuse resources as much as possible, for example by assigning the same person to all three tasks in B and C. We, however, would like to have an objective function that tells which schedules are the best, according to some optimization criteria. When estimations of the durations of tasks are available, these criteria can be ones that have been studied extensively in scheduling literature, like maximum completion time or sum of completion times. For now, we want to focus on the quality with which a resource can

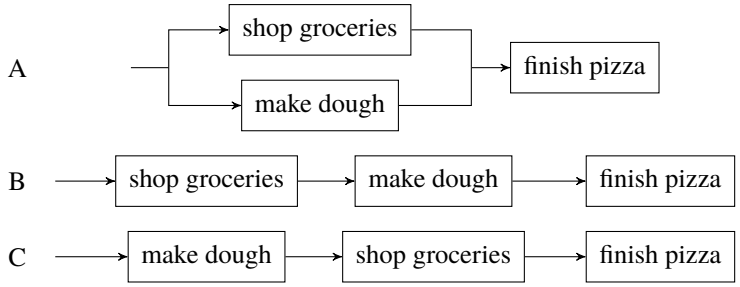


Figure 7.2: Three ways of making pizza

execute a task, rather than durations. In this example we choose workflow A, with Mom and the car as best suited for shopping, while Dad prepares the dough and finishes the pizza, because he has the most experience with these tasks.

7.5.4 Execution

With responsibilities settled, the family begins execution of the tasks. Mom drives with the car to the supermarket, while Dad starts making the dough. On the way to the supermarket, the car breaks down. In our system, this is modelled by marking both Mom and the car as degraded. Now that the goal of having pizza for dinner can no longer be reached, alternative solutions must be found. The task *shop groceries* must be stopped, as it does not make sense to see it as a task that can be taken over by other resources. This frees the resources Mom and car, but as they are degraded, they will not be considered when looking for alternatives. When starting a new instance of *shop groceries*, the system offers two possible assignments: Alice with the bike, or Bob with the bike. However, we still have Mom waiting in the broken car, and getting her home has higher priority than making pizza. Dad knows how to get the car running again, because he has repaired the same problem before.

The solution to the problem looks like this: Dad grabs his tools and takes the bus to repair the car. Alice takes his place in the kitchen and continues preparing the dough where Dad left off. Bob takes his bike and shops groceries. To implement this solution in our system, the system must grant a supervisor the following possibilities.

1. The resources Mom and car must be marked as degraded.
2. The task *shop groceries* must be cancelled.
3. A new task *repair car* must be created. This task has not been conceived in the planning phase, so the supervisor must have the ability to create new tasks on the fly.
4. The task *make dough* must be paused, so that Dad becomes free for the repair task.
5. Alice is assigned to *make dough*, and the task can be resumed.
6. A new instance of *shop groceries* is started, with Bob and the bike as resources.

The supervisor must perform steps 1 to 3 by hand, while the system offers advice for steps 4 to 6. In particular, the system should figure out that when the high-priority task *repair car* pops up, there still is a solution to make pizza.

7.6 Related Work

Kuhn [1955] showed that the assignment problem can be solved in polynomial time with the Hungarian method. The assignment problem asks to distribute n people to n jobs, where each person is differently qualified for each job, such that the total quality is maximized. Our assignment problem is a bit different however, because in general we require multiple resources per task, and when tasks run in sequence the same resources can be assigned to different tasks.

The literature on machine scheduling has results about machines that can process different kinds of jobs, which are called *multi-purpose machines*. It appears that our concept of resources can be seen as multi-purpose machines. Scheduling with multi-purpose machines and arbitrary processing times is NP-hard [Brucker et al., 1997].

Task that require multiple resources seem similar to what is known as *multiprocessor tasks* in the scheduling literature. Scheduling tasks that require two dedicated processors is already strongly NP-complete [Kubale, 1987].

Seeing that we have a combination of the assignment problem with multi-purpose machines and multiprocessor tasks, it appears that we have to resort to heuristic methods to find solutions. A promising method has been described by Mencía et al. [2015], using genetic algorithms to find schedules for tasks with precedence and skilled operators.

7.7 Future Work

The two most important next steps are further literature study and a minimal viable prototype implementation. We already know that finding exact solutions to our scheduling problem is infeasible, so we have to see what kind of techniques exist for heuristically finding okay solutions. After hopefully finding such a technique, and most probably adapting it, we have to make an implementation which allows us to run example scenarios. This lets us verify that we are on the right track, that the data model covers the relevant aspects of our domain, and that the answers we get are interesting and have the potential to become useful.

A user-friendly interface is not a goal for the prototype. In subsequent work it would be interesting to study how the differences between qualitatively similar solutions can be highlighted to the user. This is especially interesting when the choice of resources doesn't matter, for example when permutations of a set of resources all lead to solutions with identical quality. Furthermore we would like to find out how to present the consequences of choosing a particular solution for the remaining tasks.

7.8 Conclusion

In this chapter we study the problem of resource allocation in command and control scenarios. We narrow down the kind of problem we want to solve to get an understanding of how a first step towards an integrated command and control system could look like. We identify what kind of information such a system needs in order to provide useful answers. The central components in our system are *tasks* and *resources*, and their connection via *capabilities*. Tasks require capabilities while resources provide capabilities. An abstraction

we call *systems* can be used to express that a capability can only be provided jointly by several resources. Every system has a *capability function* that indicates the level of quality with which a given set of resources provides the system's capability.

Given a scenario with tasks, resources and systems, we want to solve a combination of an assignment and a scheduling problem. We want to find an ordering of the tasks together with an assignment of resources to the tasks such that the overall quality is as high as possible.

One goal of identifying the problem we want to solve is comparison with the scheduling literature. It turns out that the complexity of our problem makes exact optimization impossible. The best we can hope for is to find some form of heuristic that gives us a feasible solution if one exists and points out conflicts if no solution exists. This is acceptable, because in a realistic scenario, quickly finding some okay solution that gets the job done is more important than eventually finding the optimal solution.

Acknowledgements

We would like to thank the Manning and Automation Team at TNO, Rinus Plasmeijer, Bas van der Eng, and Terry Stroup for many hours of fruitful discussion.

8 Resource Scheduling with Computable Constraints for Maritime Command and Control

Maritime command and control currently trends towards reduced crew size with highly trained crew members, with the intention of allowing more flexible deployment. As people no longer have fixed roles, dynamic scheduling of personnel and equipment is required. In this chapter we identify what kind of scheduling problem arises for this, and how to solve it. We focus on the requirements of incident response and damage control scenarios, which can be anticipated but are impossible to plan out in advance. We perform a literature study to put our scheduling problem in context and present an algorithm that satisfies the identified requirements.

One particular requirement we want to support is the assessment of schedules with user-defined, arbitrary computable quality metrics. The estimation of how good a resource, be it a person or a machine, will utilize its capability for a given task should consider factors like weather, positions of resources, or equipment degradation. This should go alongside classical metrics like length of a schedule's critical path.

Our work should be seen as a puzzle piece for the development of an integrated mission support environment.

8.1 Introduction

In this chapter we study scheduling for command and control (C2). C2 refers to systems, processes and best practices required to coordinate people and machines to cooperatively reach a common goal. C2 is especially needed for dynamic situations where fixed business processes would not be able to deal with unexpected events. Our focus lies on the operation of navy ships, but the results should be general enough to be applicable to any kind of incident response or search and rescue operation.

This chapter is written for participants of the MAST conference, who have operational expertise, but not necessarily knowledge of automated scheduling and multi-criteria optimization.

This chapter makes the following contributions.

- We identify the kind of scheduling problem that arises in C2.
- We implement a scheduling algorithm that solves instances of the identified problem.
- We demonstrate the features and limitations of our scheduler by means of example scenarios.
- We perform a literature study to compare our problem with related work.

This chapter combines known concepts of the fields of project scheduling, multi-criteria optimization and evolutionary algorithms and applies them to a novel variant of the resource-constrained project scheduling problem.

The long-term goal of this line of research is to develop methods for decision making support. Staying ahead of hectic situations requires two sorts of skills. On the one hand, it requires flexible and creative out-of-the-box thinking, something that humans are good at. On the other hand, it requires keeping track of large amounts of minute details, and quick assessment of different courses of actions, something that computers are good at. Current state-of-the-art in the Royal Netherlands Navy is that humans do most of the task coordination by hand, with rudimentary or no computer support. The potential for improvement, even with simple tools, is immense. Decision making support provides tools for human-machine collaboration where each party can contribute their respective strengths.

Focus. Carrying out missions involves a constant cycle of decision making and acting, on several levels of scale. There is a large cycle between individual missions of planning, execution, and assessment. Inside of each mission, there is a much tighter, and constantly running, variant of the same process. A common model for this cycle is the OODA loop [Boyd, 1976], which stands for Observe-Orient-Decide-Act. The *decide* part can be further subdivided into two halves. The first half is about deciding which actions should be taken in a given situation, while the latter half is about who carries out the actions and when. This division is advocated by Bolderheij [2007]. Our work focuses on the latter half. In particular we see our work applicable in damage control situations, where workload spikes and degraded resources cause resource conflicts. We do not focus on the happy flow of missions, which by design has no resource conflicts.

8.1.1 Terminology

A *task* is work in the real world, the execution of which should be managed by our system. A *basic task* is an atomic unit of work. Atomic means that it is not useful to divide a task into subtasks. For example, repairing a machine is a basic task. A mechanic only needs to be told that a machine needs repairing, and when the best time is to start. The mechanic can then autonomously carry out the repair, even though conceptually it consists of many tiny subtasks. There would be no benefit in letting our system manage these subtasks. The decision which tasks are basic tasks and which can be split up into subtasks must be made on a case-by-case basis, depending on whether one expects a benefit from from our system managing the subtasks.

A *resource* is everything and everybody required to execute a task. To execute a task, all required resources must be assigned to it. We use the word “assigned” symmetrically: We say a resource is assigned to a task, but also that the task is assigned to the resource. Examples for resources are fire fighters and fire extinguishers, mechanics and tool boxes, or radars and other sensors. The fire to be extinguished, the machine to be repaired and the contact to be tracked are not resources, but *cases*. Sometimes there is overlap between resources and cases. This becomes a problem for our assumption that planning and scheduling can be clearly separated, as will be discussed in section 8.2.1.

Resources come in two varieties: *consumable* and *reusable*. Consumable resources are used up when executing their assigned task. Reusable resources become free again when

the task is finished, and can be assigned to new tasks. Examples for consumable resources are fuel and food, examples for reusable resources are tools and people. In earlier work [Klinik et al., 2017b,a] we study this difference in more detail. For simplicity we focus entirely on reusable resources in this chapter.

The connection between tasks and resources are made with *capabilities*. Resources have one or more capabilities, and tasks have one or more *capability requirements*. For example, Bob has the capabilities Operator, Medic, and Commander. The task Perform rescue requires a Transport and an Operator. From a technical standpoint, capabilities are just labels that the scheduler uses to match resources and tasks. Because the scheduler does not care what capabilities actually stand for, they can be used in different ways. For example, capabilities can represent professions, completed trainings, military ranks, or identify individual resources. In the scheduling literature capabilities are often called *skills*. We chose to avoid this word because it suggests that resources are people. We want to emphasize that resources can be anything including machines, people, and tools, so we chose a more general term.

8.2 Scheduling

Scheduling is the activity of determining, given a set of tasks to be executed, who executes the tasks and when. A large amount of different scheduling problems have been studied in the literature [Dürr et al., 2016]. A *scheduling problem* describes the characteristics of the problem to be solved. For example, job-shop machine scheduling problems have the characteristic that one task needs exactly one machine to be executed. In contrast, project scheduling problems involve tasks that require several resources simultaneously. An *instance* of a scheduling problem is a set of concrete tasks and resources with concrete numbers for which a schedule has to be found. A *scheduler* has the responsibility of finding a schedule given an instance.

8.2.1 Planning Versus Scheduling

In the literature on automated planning and scheduling, for example Ghallab et al. [2004], planning and scheduling are treated as two distinct problems that can be solved in isolation. Planning takes as input a start state, a set of state-transforming actions, and a goal predicate. The purpose of planning is to choose actions and chain them together to transform the start state into a state that satisfies the goal predicate, via a series of intermediate states. Such a chain of actions is called a *plan*. Planning is only concerned with the effects of actions on the state, but not with who should execute them and when. This is the purpose of the scheduler. The output of the planner is the input of the scheduler. In other words, planning answers the question “What to do?”, while scheduling answers the question “Who will do it?” There are two problems with this division.

Firstly, the output of state-of-the-art planning algorithms is a *list* of actions, which imposes a total order on the actions. In most situations in reality however, only some actions of a plan have an inherent ordering constraint. For some actions the order in which they are executed does not matter. Scheduling works best if it gets as input a partial order, so that the scheduler has some freedom to rearrange and parallelize tasks. Planning in

which the output is a partial order instead of a total order is called *partial order planning*, and is an active field of research, see for example the work by Muise et al. [2016]. For this chapter, we assume that a partial order of tasks is given, regardless of how it is obtained.

Secondly, there are situations in which the decision *who* executes a task influences *which* other tasks must be executed. In other words, scheduling can influence planning. This contradicts the assumption that planning and scheduling can be clearly separated. This arises typically in situations where a resource has to be prepared in order to be used. For example, when there is a helicopter and a boat for a mission, then choosing one might require different preparation tasks, or a different approach to the mission altogether. In other words, the mission cannot be planned before a resource is selected. Taking such situations into account would complicate our algorithm substantially, leading too far away from our current goal of scheduling resources. We therefore make the assumption that planning and scheduling *can* be separated. It is left to the users of our system to model their problems in such a way that resource selection does not influence planning. See section 8.2.3 for an example.

8.2.2 Scheduling for C2

The scheduling problem that arises in C2 is a variant of the multi-skill resource-constrained project scheduling problem (MSRCPSP). The MSRCPSP [Artigues, 2008] is a variant of the resource-constrained project scheduling problem (RCPSP), which is itself a variant of the project scheduling problem (PSP). In the basic PSP, each task requires specific resources to be executed. For example, the task *review customer complaint* specifically requires Alice and Bob. The goal is to find a time slot where both Alice and Bob are available.

The RCPSP generalizes the PSP in that each task can require a quantity of one or more of a class of resources. Every resource can only belong to exactly one class. For every class there is a pool of available resources. For example, it is possible to express that the task *review customer complaint* requires one PR manager and two engineers. The goal is to find a PR manager and two engineers and a time slot such that all people are available during that time.

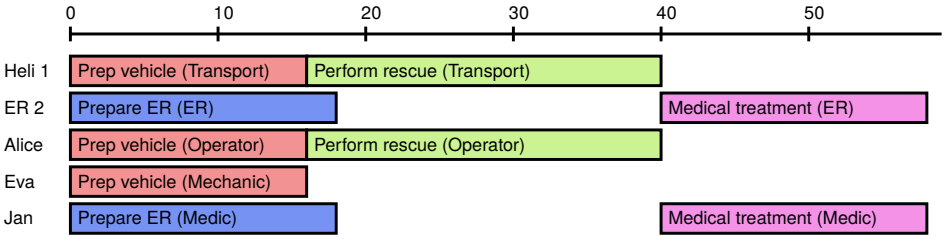
In the MSRCPSP, resources can belong to multiple classes, which are called *skills* in the literature. A resource can have multiple skills, for example Bob can be a PR manager and a technical editor. When Bob is chosen as PR manager for a task, he is also no longer available as a technical editor. This means that resources can no longer be grouped into pools where a number describes how many resources are available in a pool. This makes the MSRCPSP harder to solve than the RCPSP.

The MSRCPSP is almost what we want for our maritime setting, especially the direction the Royal Netherlands Navy wants to pursue. We need to extend the MSRCPSP in two ways however to make it applicable to our scenarios. These extensions are motivated by an example and described in the following sections.

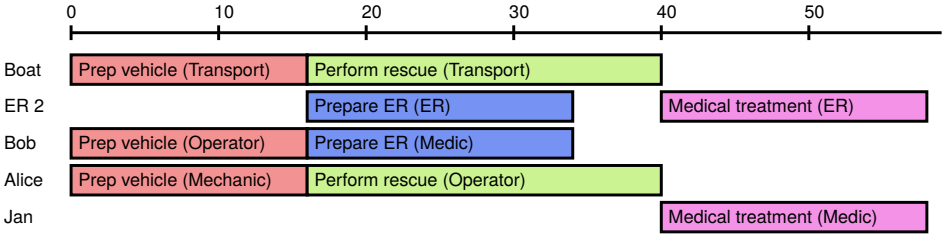
8.2.3 Example Scenario: Search and Rescue

Consider the following example, in which the coast guard has to perform a search and rescue operation. To perform this operation, four tasks must be carried out:

- Prepare transport, which requires a transport, an operator, and a mechanic.



(a) In good weather the helicopter can be used



(b) In bad weather the boat must be used

Figure 8.1: Two possible schedules for the search and rescue scenario

- Prepare ER, which requires an emergency room and a medic.
- Perform rescue, which requires a transport and an operator.
- Give medical treatment, which requires an emergency room and a medic.

The coast guard has three transports: Two helicopters and a boat. There are two emergency rooms. There are four people with the following skills: Bob is a medic and an operator. Alice is a mechanic and an operator. Eva is a medic and a mechanic. Jan is a medic. The tasks partially depend on each other. Perform rescue depends on Prepare transport. Give medical treatment depends on both Prepare ER and Perform rescue. Figure 8.1 shows two possible schedules for this scenario.

8.2.4 Resource Affinity

In the MSRCPSP, tasks specify the skills of the resources they require. The scheduler is free to pick any resource that has the required skill. In our search and rescue example there are two places where there is an additional restriction, not expressible in the MSRCPSP. The tasks Prepare transport and Perform rescue both require the same transport. It does not matter which transport is selected, but the same must be selected for both tasks. The same is true for the emergency room for the tasks Prepare ER and Give medical treatment. The operator for Prepare transport and Perform rescue may be different, as may be the medic for Prepare ER and Give medical treatment. In a C2 scheduling problem it should be possible to express the constraint that the same resource must be chosen for two capability requirements of different tasks. We call such constraints *resource affinity constraints*. Both schedules in fig. 8.1 have the same transport and the same emergency room selected, but the scheduler is free to select different operators and different medics.

8.2.5 Arbitrary Computable Quality

Another extension of the MSRCPSP required for C2 is that the quality with which a resource can utilize its capability for a task can depend on external circumstances. In our example we assume that a helicopter can only be deployed in good weather. When the weather is bad, the boat must be used. Additionally, Bob does not have a helicopter license, so if the helicopter is selected, Bob cannot be selected as operator. The schedule in fig. 8.1a has been calculated in a good weather situation. A helicopter is selected as transport, but Bob is not the operator. The schedule in fig. 8.1b has been calculated in a bad weather situation. The boat is selected as transport, and Bob as operator.

In general, we want the quality of resources for tasks to be arbitrary computations. One can think of things like the distance of a person to a room on fire, the type of a fire, the hours since the last time a person rested, or rules of engagement. This is in contrast to the treatment of the MSRCPSP in other works. For example in Myszkowski et al. [2018], every resource has a fixed cost, called a *salary*. No matter under which circumstances to which task a resource is assigned, its cost is always salary multiplied by task duration.

Quality can depend on the state of the outside world. The state of the world is kept in a data store called the operational picture (OP). In reality, the OP is an estimation about the state of the world, based on interpretations of various data sources like radars, sensors, or network connections. As such the OP contains mistakes, uncertainties, or might be out of date. How to deal with uncertain information is not the focus of this chapter. We make the assumption that the OP is accurate enough to allow being taken at face value. For our purpose the OP contains both external and internal factors, like positions of contacts and weather conditions, or crew availability and equipment degradation. In our system the OP is a simple collection of key-value pairs, but in reality it can be a combination of several databases. The important point is that we enable the function that determines the quality of a resource for a task to take the OP into account.

Given the above considerations, we conclude that quality functions must be part of problem instances. This means that the user can define a formula that takes multiple factors into account to determine how qualified a resource is for a task. Quality functions take the following parameters.

- The **resource** whose capability should be assessed. For example Bob.
- The **task** to which the resource is assigned. For example Prepare transport.
- The **capability requirement** to which the resource is assigned. The quality function needs to know whether Bob is assigned as operator or mechanic to Prepare transport.
- The **operational picture**. If the OP says that Bob is at the end of a 10 hour shift, he might perform with decreased quality.
- The **decision vector** that tells which resources are assigned to the other capability requirements. If the helicopter is assigned as transport to Prepare transport, Bob performs badly as operator.

For example, when we want to track a contact with a radar, the quality function can take the type of the contact and the weather into account to return the estimated tracking error in meters. When there are several radars available, we would like to pick the one with the smallest error. When a person has to extinguish a fire, the quality function can return the time in seconds it takes for the person to reach the room. When Bob has to be

the operator for Perform rescue, the quality function can return a unitless number 1 or 2, depending on whether he would have to operate a helicopter or a boat. These examples illustrate that the unit of the value that a quality function returns can differ depending on the combination of task and capability requirement. It is therefore not useful to directly compare outcomes of quality functions of different tasks with each other, only the outcomes of the same capability requirement for different resources. Combining all objectives so that schedules can be compared is the topic of section 8.4.

We call the result of a quality function an *objective*. This is a general concept with which we try to handle quality assessments of different units. Some objectives are better when they are maximized, some are better when they are minimized. An objective is therefore a number together with a tag Maximize or Minimize which indicates how to compare two objectives of this type. Quality functions are not the only source of objectives in a C2 scheduling problem. Another objective is the makespan, which is the overall time from the start of the first task to the end of the last task. The makespan should be minimized.

Every problem instance brings its own quality function. To evaluate a schedule, the quality function is applied to every capability requirement that has a resource assigned to it. This results in a vector of objectives. For example, the schedules in fig. 8.1 belong to an instance where nine capability requirements must be assigned. These assignments are called the *decision vector* of a solution, because the scheduler must make nine assignment decisions. The quality function is applied to nine capability requirement/resource pairs. Together with the makespan, every decision vector gives rise to a total of ten objectives, which is called the *objective vector*. The goal of the scheduler is to find a decision vector that has a good objective vector. The difficulties of comparing objective vectors are described in section 8.4.

8.3 Definition of the C2 Scheduling Problem

With all the previous considerations in mind, we now define the C2 scheduling problem. In addition to what was said before, we make the following assumptions. No preemption: Tasks cannot be interrupted once they are started. Task durations are integer: Durations are unitless numbers which can stand for days, hours, or minutes. Fractions are not necessary. Capability requirements are known in advance and do not change while tasks are being executed.

Let C be a set of capabilities. Let T and R be sets of task and resource identifiers, a unique one for every task and every resource.

Tasks A task is a tuple $\langle t, d, \vec{c}, \vec{p} \rangle$ where t is a unique task identifier, $d \in \mathbb{N}$ is the task duration, $\vec{c}$ is a list of capability requirements, and $\vec{p}$ is a list of predecessor task identifiers.

Resources A resource is a tuple $\langle r, \vec{c} \rangle$, where r is a unique resource identifier, and $\vec{c}$ is the list of capabilities of the resource.

Resource affinity constraints A resource affinity constraint is an equality of the following form.

$$\langle t, i \rangle = \langle u, j \rangle = \ldots$$

The meaning of such an equality is that the resource assigned to capability requirement c_i of task t must be the same as the one assigned to capability requirement c_j of task u, and so on. We require that all entries in a resource affinity constraint refer to the same capability.

Assignments An assignment A is a set of tuples $\langle t, i, r\rangle$, one tuple for each capability requirement of each task, that assigns resource r to capability requirement c_i of task t.

Schedules A schedule is a tuple $\langle A, S\rangle$ of an assignment A together with a set S of start times $\langle t, s\rangle$ that assigns a start time $s \in \mathbb{N}$ to every task t.

Validity We say that an assignment is *valid* iff

- all resources assigned to the same task are different
- the assignment respects the resource affinity constraints.

We say that a schedule is *valid* iff

- its assignment is valid
- it respects the task ordering, which means every task's start time s_t is after the end time of all of its predecessors $\vec{p}_t$
- no resource is assigned to two tasks that are overlapping in time.

The end time of a task is its start time plus its duration.

Quality functions A quality function $f(r, t, i, E, A)$ takes as arguments a resource $r \in R$, a task $t \in T$, a capability index i, an environment E, and an assignment A, and returns either Maximize q or Minimize q with $q \in \mathbb{R}$. The quality function calculates how well r will perform capability requirement c_i of task t, given an environment and an assignment. Maximize and Minimize are annotations that specify if q is a more-is-better or less-is-better objective. We require that quality functions are consistent with these tags, which means for each t and c_i it returns the same tag. Only then is it meaningful to compare different resources for the same capability requirement of a task.

C2 scheduling problem With the definitions above we can now define the C2 scheduling problem. Given a set of capabilities C, a set of tasks T, a set of resources R, a set of affinity constraints, an environment E, and a quality function f, calculate a valid schedule $\langle A, S\rangle$ that optimizes the objective vector

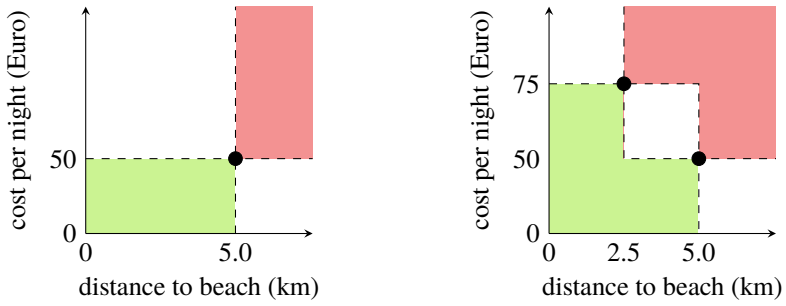
$$\langle f(r, t, i, E, A)\rangle \text{ for each } \langle r, t, i\rangle \in \text{A}$$

Optimizing the objective vector means to find the largest possible q for each Maximize q, and the smallest possible q for each Minimize q.

8.4 Multi-Criteria Decision Making

Multi-criteria decision making (MCDM) is the problem of picking a preferred solution from a set of candidates that performs best in a number of conflicting criteria. This section gives a brief introduction to the topic. More information can be found in the book by Keeney and Raiffa [1993].

In our domain, the decision consists of a number of sub-decisions: one has to decide which resource to assign to each capability requirement. The decisions can be represented by a decision vector $\vec{x} = \langle x_1, x_2, \ldots, x_n\rangle$, where each x_i stands for one capability requirement of a task. Each decision variable x_i has its own domain. For example, the decision variable



(a) One candidate

(b) Two candidates

Figure 8.2: Two incomparable hotels

Perform rescue (Transport) has domain { Heli 1, Heli 2, Boat }, while the decision variable *Perform rescue (Operator)* has domain { Bob, Alice }. Every decision vector gives rise to an objective vector $\vec{q} = \langle q_1, q_2, \ldots, q_m \rangle$. The lengths of $\vec{x}$ and $\vec{q}$ can be different, because each decision can lead to multiple objectives.

To understand why MCDM is difficult, consider a much simpler problem. Imagine we are planning a seaside vacation, for which we are looking for a hotel. We want the hotel to be as close to the sea as possible and as cheap as possible. For the sake of this example, these are the only two criteria of interest. There is only one decision variable x, whose domain is the set of hotels we know of so far. Every hotel has an objective vector with two elements $\langle q_1, q_2 \rangle$, the distance to the sea and the price, both of which should be minimized. As we discover new hotels, we want to compare them to the ones we already know of. If a hotel is both cheaper and closer to the sea, it is definitely better according to our criteria. If a hotel is both more expensive and further from the sea, it is definitely worse. But what if we find a hotel that is closer to the sea but more expensive, or the other way around? This is the central problem of MCDM: *It is not possible to compare objective vectors that are better in some and worse in other components*. The result of MCDM is therefore usually a set of candidates that are pairwise incomparable. These are called the *non-dominated candidates*. If we have exhaustively discovered all hotels at the desired destination, this set is called the *Pareto front*.

Figure 8.2 illustrates the issue. Figure 8.2a shows one hotel with a distance of 5.0 km from the sea that costs 50 EUR per night. Discovering this hotel divides the criterion space into four areas. The red area to the upper-right marks hotels that are definitely worse according to our criteria. Should we discover a hotel that falls into this area, we can immediately disregard it, because the current one *dominates* it. The green area to the lower-left marks hotels that are better in both criteria. Should we discover a hotel that falls into this area, we would immediately discard the current one, as it is *dominated by* the new one. The white areas to the upper-left and lower-right mark hotels that are closer but more expensive and cheaper but further away, respectively. Figure 8.2b shows what happens when we discover a hotel that falls into a white area. We have to remember it as a *non-dominated* candidate, and the shape of the red and green areas changes. Once we have discovered all hotels in the area, we end up with a set of non-dominated candidates.

Without further information about the candidates and personal preferences of the decision maker, it is not possible to narrow down the selection.

8.4.1 Scalarization

In the previous section we discussed that two objective vectors are incomparable if each is better than the other in at least one objective. Deb [2011] writes that in situations where objective vectors are longer than three, most candidates tend to become incomparable. The example scenario in section 8.2.3 has objective vectors of length 10, and realistic scenarios can easily have 50 or more objectives. This means that any pure MCDM method will present the user with large and diverse sets of solutions.

To give the user more useful answers, these solutions should be further ranked and evaluated. This can be done by scalarizing their objective vectors, which means all objectives are combined into a single number. This process is called *scalarization*. Scalarization discards information and must be done carefully to not dismiss favourable solutions. Two common scalarization methods are the weighted-sum- and the weighted-product method. Tofallis [2014] highlights the disadvantages of the weighted-sum method and advocates using the weighted-product method. He argues that the weighted-sum method is unintuitive and error-prone to use. The weighted-product method is easier to use and can capture user preferences more faithfully.

The weighted-product method works as follows. Let $\vec{q} = \langle p_1, p_2, \dots, q_1, q_2, \dots \rangle$ be an objective vector where the p_i are more-is-better objectives and the q_i are less-is-better objectives. Let $\vec{w} = \langle w_1, w_2, \dots, v_1, v_2, \dots \rangle$ be the weights, pre-determined by the decision maker. The score s of an objective vector is calculated using the weighted product according to the following formula.

$$s = (p_1^{w_1} p_2^{w_2} \dots) / (q_1^{v_2} q_2^{v_2} \dots)$$

As less-is-better objectives occur in the denominator, they cannot be zero. If a more-is-better objective is zero, the whole score will be zero, which might not be desirable. Programmers of quality functions need to take both into account. Some of the properties of the weighted-product method relevant for c2 scheduling are as follows.

The weighted-product method allows mixed units of measurement. Our main reason for choosing the weighted-product method is that the units and magnitudes of the individual objectives does not matter. No conversion or normalization is required, the numbers can be multiplied as-is. One objective can be a distance in meters, another one a cost in euros, yet another one a time in seconds.

Weights allow modelling diminishing returns. If an objective with weight 1 doubles, the score doubles as well. The influence of an objective on the score can be amplified by giving it a weight greater than 1. Weights smaller than 1 can be used to model diminishing returns, which is the effect that people tend to give less value to further increases in an objective. A person might be very happy about the first million they earn, but more money does not make them equally more happy. As a famous quote by Arnold Schwarzenegger goes: “I now have 50 million but I’m just as happy as when I had 48 million.” For example, when an objective with weight 0.5 doubles, the score increases by a factor of about 1.4, but if it quadruples, the score only doubles. The effect of a weight q^w on the overall score can be estimated as follows. If the objective q changes by 1%, the score approximately

changes by $w\%$. For a derivation of this estimation, see Tofallis [2014]. Section 8.7 has an example that demonstrates how different weights can guide our algorithm towards different solutions.

To utilize the weighted-product method, all objectives must be units of ratio scale. For a unit of measurement to be of ratio scale, it must have a meaningful zero value that represents absence of the measured quantity. Furthermore it must allow multiplication by positive constants. This allows comparisons based on ratios of measurements, like ”this is twice as long as that”. Examples are mass, length, time, angle, money, percentage, and temperature in Kelvin. Temperature in degrees Celsius is not of ratio scale, because it does not have a meaningful zero value. 20°C is not twice as hot as 10°C. Ratio scale is a level of measurement as described by Stevens [1951]. A more recent exposition is given by Wohlin et al. [2012].

8.5 Evolutionary Algorithms

We chose to use an evolutionary algorithm to calculate schedules. This section describes what evolutionary algorithms are and why they are suited for our problem.

Evolutionary algorithms are a class of probabilistic population-based meta-heuristics for solving optimization problems. *Probabilistic* means that pseudo-random number generation is used to guide the algorithm. There is no guarantee that the algorithm will always find the best solution, nor that two runs of the same problem give the same result. *Population-based* means that the algorithm has a set of candidate solutions which it tries to improve step by step. The result is a set of solutions. A *meta-heuristic* is an algorithm that performs optimization without knowing about the underlying problem. This is in contrast to normal heuristics, which use knowledge about the problem being solved. A heuristic always guides a search algorithm through a search space towards an optimal solution. For example, when searching for the shortest path on a map, the A^* search algorithm can use as heuristic the straight-line distance to the goal to determine which path to extend next. The straight-line distance is a problem-specific heuristic that can only be used in Euclidean spaces, but it works well for that purpose. This is what is meant by a heuristic using knowledge about the problem.

Evolutionary algorithms fit our problem nicely, for two reasons. First we are interested in alternative solutions, not just one solution. Our algorithm calculates a final population, which is the set of the best solutions it has discovered, ranked by the quality function. Second, user-defined quality functions make the search space unpredictable and non-continuous. Because quality functions are Turing-complete calculations, it is impossible to estimate what kind of changes to a candidate will make it better or worse. Meta-heuristics do not need knowledge about the function they are trying to optimize, which makes them a good fit for user-defined quality functions. Deb [2001, 2011] has conducted a lot of research about using evolutionary algorithms for multi-objective optimization and decision making. His work has been a big inspiration for this chapter.

8.5.1 Ingredients for Evolutionary Algorithms

There are many variants of evolutionary algorithms, but they all need the following parts.

Chromosomes A chromosome is a list of bits, or a list of numbers in our case, that encodes a solution. A population is a set of chromosomes, and an evolutionary algorithm tries to construct chromosomes that encode good solutions.

Decoder A decoder is a function that takes a chromosome and calculates a solution from it, in our case a schedule. The evolutionary algorithm uses the decoder to compare and rank the chromosomes of a population.

Evaluator An evaluator is a function that compares two solutions. The evolutionary algorithm uses the evaluator to rank the chromosomes of a population in order to determine which ones to use for the construction of the next population.

Crossover and mutation Crossover and mutation are two functions that directly manipulate chromosomes. Many standard variants are possible, but they can also be provided on a problem-specific basis. Crossover gets two parent chromosomes as input and produces one or more child chromosomes, with the hope that the children have some good properties from both parents. Mutation randomly changes one entry of a chromosome. In some encoding schemes, not all possible chromosomes encode valid solutions. Crossover and mutation should be chosen such that when given valid chromosomes, they produce valid chromosomes.

Selection strategy Genetic algorithms work by constructing new populations from old ones. To do this, the algorithm has to select chromosomes from the old population to which to apply crossover and mutation. In general it is a bad idea to just select the best chromosomes of a population, as this can lead to fast convergence towards a local optimum. Populations should be diverse and include non-optimal solutions from distant parts of the search space, in order to discover good solutions far away from each other. To preserve diversity, the algorithm should select a mix of good and bad chromosomes to produce offspring.

The evolutionary algorithm With these ingredients, an evolutionary works as follows. It first generates an initial population of chromosomes randomly. Then, until the stop criterion is reached, it calculates new populations. For this, the chromosomes of the current population are decoded, evaluated, and sorted from best to worst. The selection strategy picks those chromosomes that are allowed to produce offspring. The offspring is decoded and evaluated, and sorted into the current population. The worst chromosomes are discarded, so that a new population of the same size as the old population is obtained. There are different possible stop criteria, the simplest one being a fixed number of generations. Another one is convergence, which stops when the best solution has not changed for a fixed number of generations.

8.5.2 Criticism of Evolutionary Algorithms

By their nature, probabilistic algorithms can never guarantee convergence to the best solution, if it exists. Articles about evolutionary algorithms usually use a number of computer experiments to obtain performance data which is then compared to estimated best solutions. In the literature on classical algorithms, it is customary to provide correctness proofs for algorithms. No such proof can be given for probabilistic algorithms. The best we can do is guarantee that populations do not get worse over time, by always keeping the best solution found so far.

Another problem with evolutionary algorithms is that their effectiveness depends on the

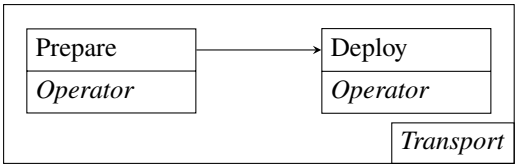


Figure 8.3: A resource frame in a simple prepare-deploy scenario. The resource requirement of the frame applies to all tasks inside the frame.

input parameters, like chosen selection strategy, population size and mutation probability. Which parameters work well can change with the problem instance, and may require some adjustments until satisfying results can be reliably produced. End users, who are only interested in the results and not the inner workings of the algorithm should not be bothered with such technicalities. We need further research and experiments with realistic scenarios to develop a parameter set that performs reasonably reliable in practice for our problem.

8.6 Implementation

In this section we describe some design decisions of the implementation of our method. We have implemented our scheduling algorithm in the functional programming language Clean. The source code is available online [Klinik, 2019]. The scheduler works in two phases. Phase one uses a genetic algorithm to find a set of good assignments for a given instance definition. Phase two uses these assignments to build schedules.

8.6.1 Instance Definition

A C2 instance definition consists of a list of resource definitions, a list of task definitions, a list of resource frame definitions, a list of weights, and a quality function. Resources, tasks, and frames are numbered according to the order in which they appear in the definition.

A resource definition consists of a name and a list of skills. A task definition consists of a name, a duration, a list of skill requirements, and a list of predecessors. A weight is a floating point number.

Resource frames specify resource affinity constraints. A resource frame has a list of tasks and a list of resource requirements. Every resource requirement of a resource frame is by definition a resource requirement of all tasks inside the frame, and assigning a resource to a frame automatically assigns it to all tasks in the frame. See fig. 8.3 for an example. There are two tasks, Prepare and Deploy, which both require an Operator. The tasks are inside a resource frame that requires a Transport. This means that both tasks require a Transport, and that the transport assigned to both tasks must be the same one.

8.6.2 Genetic Algorithm

Our genetic algorithm is based on the implementation by Alexeev [2014]. We ported his code from Haskell to Clean and extended it in a number of ways to make it fit our needs.

The algorithm has four input parameters: the problem instance, the population size, the mutation probability, and the number of generations.

Encoding. Our chromosome encoding scheme is based on the one by Myszkowski et al. [2018], extended with support for tasks with multiple skill requirements. A chromosome corresponds to a decision vector, and encodes an assignment as follows. A chromosome is a list of numbers, one number for each resource requirement in a resource definition. The domain of an entry in a chromosome is the list of all resources that have the corresponding capability. For example, the instance in fig. 8.3 has three capability requirements in total, one of each task and one of the frame. Let there be four resources: Alice and Bob, who are both Operators, and a Heli and a Boat which are both Transports. A chromosome for this instance is a list of three numbers $[o_p, o_d, t]$. The first two numbers o_p, o_d stand for the Operator of the Prepare and Deploy tasks, respectively. They have the domain $[Alice, Bob]$. The third number stands for the transport, and has domain $[Heli, Boat]$. For example the chromosome $[0, 1, 1]$ encodes the assignment of Alice as the Operator of Prepare, Bob as the Operator of Deploy, and the Boat as the Transport of both tasks.

Crossover. We use single-point crossover, resulting in two children. Single-point crossover takes two chromosomes, randomly determines a cut-off point, and produces two children by combining the first part of the first parent with the second part of the second parent, and vice versa. The following table illustrates this scheme with parents A and B.

Parents	Children
$[A, A, A, A, A, A]$	$[A, A, B, B, B, B]$
$[B, B, B, B, B, B]$	$[B, B, A, A, A, A]$

Mutation. Our mutation operation randomly determines a location in a given chromosome and changes it to a randomly selected entry in the corresponding domain.

Selection. We use the selection method of Alexeev [2014], where all chromosomes in a population are paired with each other. This makes the algorithm run with quadratic complexity in the population size, so it is advised to use reasonably small populations. The children of crossover are subjected to mutation with the given mutation probability. Alexeev discards the old population and continues with the new one. We changed this so that the old and new populations are combined, sorted according to the fitness function, and then the best chromosomes are taken to be the new generation. This way the algorithm preserves the best individuals found so far.

Constraints and invalid assignments. There are two kinds of constraints that every assignment must satisfy. First, no resource can be assigned simultaneously to multiple capability requirements of the same task. For example, if some task requires an Operator and a Medic, and Alice has both capabilities, she cannot be assigned to both. Second, affinity constraints require the same resource to be assigned to several resource requirements of different tasks. The second kind of constraint is always satisfied by the way we implement resource frames. The algorithm selects one resource for the requirement of the frame, which applies to all tasks in the frame. The first kind of constraint can be violated, and must be checked by the fitness function. Our fitness function therefore returns either a valid score or an invalid score. A valid score is the weighted product of all qualities, while an invalid score is the number of duplicate assignments. Individuals in a population are ranked

as follows. Invalid scores are always worse than valid scores. Valid scores are ranked according to their weighted product. Invalid scores are ranked according to the number of duplicate assignments, where less is better. In this way, the algorithm can explore invalid regions of the search space, while being incentivized to avoid them in favour of valid regions.

Scalarization. The algorithm sorts individuals by comparing the weighted product of their qualities. Calculating the product of a list of numbers can result in very high numbers, which might result in overflow or loss of precision, due to the limitations of calculations on computers. We mitigate this problem by utilizing an equivalent formula of the weighted product, that directly calculates the ratio of the weighted product of two given assignments. To understand the problem, let $x = \langle p_1, p_2, p_3, q_1 \rangle$ and $y = \langle P_1, P_2, P_3, Q_1 \rangle$ be the qualities of two assignments, where p and P are more-is-better criteria, and q and Q are less-is-better. Let $\langle w_1, w_2, w_3, w_4 \rangle$ be the weights. We could calculate their scores individually and then take the ratio of the scores, as follows.

$$s_x = \frac{p_1^{w_1} p_2^{w_2} p_3^{w_3}}{q_1^{w_4}} \quad s_y = \frac{P_1^{w_1} P_2^{w_2} P_3^{w_3}}{Q_1^{w_4}} \quad r = \frac{s_x}{s_y}$$

The result r is the factor by how much better x is than y. If r is greater than 1, then the score of x is r times better than the score of y. If r is less than 1, then y is $\frac{1}{r}$ times better than x.

This calculation has the subexpression $p_1^{w_1} p_2^{w_2} p_3^{w_3}$, and similar for the P's, which can result in an intermediate result large enough to cause information loss. By using elementary laws of multiplication and division, we can alternatively calculate r as follows.

$$r = (\frac{p_1}{P_1})^{w_1} \times (\frac{p_2}{P_2})^{w_2} \times (\frac{p_3}{P_3})^{w_3} \times (\frac{Q_1}{q_1})^{w_4}$$

We know that the p_i and P_i are objectives of the same capability requirement, so they are most likely in the same order of magnitude. This results in smaller intermediate numbers, which decreases the chance for information loss.

8.6.3 Greedy Schedule Building

Our schedule builder is a modified variant of the one by Myszkowski et al. [2018]. Given a valid assignment, it builds a schedule as follows.

- Always keep track of the earliest point in time when each resource is available. Initially, this is 0 for every resource.
- For every task in the instance definition, in the order in which they occur, first schedule all predecessor tasks.
- Then, when a task has no more unscheduled predecessors, find the earliest time where all resources assigned to that task are available. This becomes the start time of the task.
- Finally, update the resource availabilities by setting them to the end time of the task.
- Repeat until all tasks are scheduled.

The difference between our schedule builder and the one by Myszkowski et al. is that they do not perform depth-first dependency resolution. In their instance definitions, all

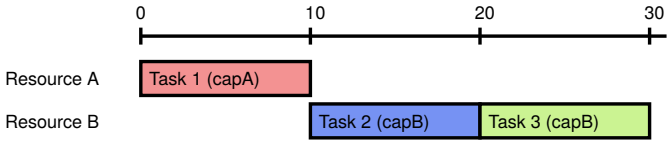


Figure 8.4: A problem instance where the greedy schedule builder fails to find the best schedule. Task 3 could be scheduled at time 0.

predecessors of a task must occur in the instance definition before the task itself. Otherwise the generated schedule may violate the ordering constraints. Our method allows tasks to occur in any order in the instance definition. Greedy schedule building, our modification notwithstanding, has some advantages and a disadvantage. The advantages are as follows.

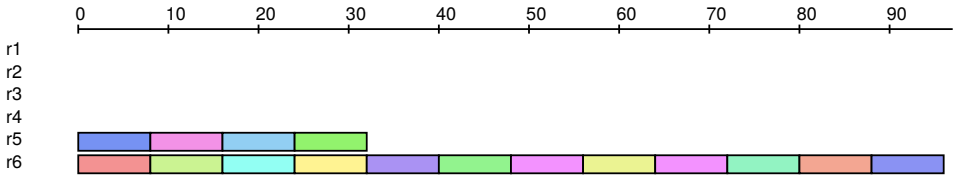
- It is simple to understand and easy to implement.
- It allows modelling task priorities, because tasks earlier in the instance definition get scheduled first.
- The schedules it builds are valid by construction, where no resource is assigned to two tasks that are overlapping in time, and where tasks are always preceded by their predecessors.
- Every valid assignment has at least one valid schedule, and the schedule builder will always find one. In the worst case, all tasks have to be executed in sequence.

The disadvantage is that the schedule builder does not always find the schedule with the shortest makespan, because it schedules the tasks in the order in which they occur in the instance definition. It is sometimes possible to find shorter schedules by changing the order of tasks. Consider the scenario in fig. 8.4. There are two resources, resource A with one capability capA, and B with one capability capB. There are the three tasks 1, 2, and 3, where Task 1 requires capA and Tasks 2 and 3 require capB. Furthermore, Task 1 is a predecessor of Task 2. There exists exactly one valid assignment, namely the one used in fig. 8.4. The greedy schedule builder processes tasks in the order in which they occur, so it fails to see that Task 3 can be scheduled at time 0. Finding the shortest schedule for a given assignment is itself an optimization problem, which we leave for future work.

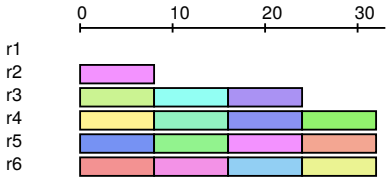
8.6.4 Quality Functions

Every problem instance comes with a quality function, as described in section 8.2.5. We want quality functions to be arbitrary algorithms in some programming language, so that users can conveniently specify them. For practical reasons we have decided for now to hard-code quality functions in the main executable. Every instance still can have its own quality function, but the main executable has to be recompiled when quality functions are added or modified. This means that quality functions must be specified in Clean, and must be registered in a lookup table so that the scheduler can find them when needed.

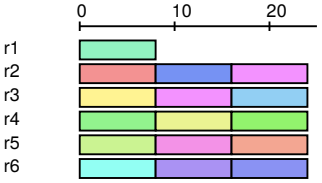
The operational picture is implemented as a JSON data structure in a text file. When the scheduler starts, it looks for such a file accompanying the instance definition. If it finds one, it loads it and passes it to every invocation of the quality function. The quality function can use the JSON query routines of the Clean standard library to access the operational picture.



(a) Makespan weight = 1.0



(b) Makespan weight = 8.0



(c) Makespan weight = 15.0

Figure 8.5: An instance to demonstrate the conflict between quality and makespan. Resource r6 has the highest quality, the scheduler prefers assigning tasks to it. As the weight of the makespan grows, the scheduler assigns more tasks to resources with lower quality.

8.7 Examples

This section demonstrates our method using two examples. The example in section 8.7.1 demonstrates the effect of weights on conflicting objectives, while the example in section 8.7.2 demonstrates how user-defined quality functions can take the location of crew members into account.

8.7.1 Conflicting Objectives

Figure 8.5 illustrates the effect of weights on conflicting objectives. This instance has six resources and 15 tasks. There is only one capability requirement C and all resources have this capability, which means all resources can execute all tasks. In this example resource 1 has quality 1 for all tasks, resource 2 has quality 2 and so on, which means that resource 6 has the highest quality. The makespan, which is the time from the start of the first task to the end of the last, is in conflict with the quality of the assigned resources. Assigning all tasks to resource 6 gives the best quality, but also the longest makespan.

Figure 8.5a shows that with equal weights, the scheduler tries to assign as many tasks as possible to resource 6. Some tasks are assigned to resource 5, which is an artefact of the probabilistic nature of our algorithm. Letting it run for a longer time eventually finds solutions where all tasks are assigned to resource 6.

Figures 8.5b and 8.5c show that as we increase the weight of the makespan, the scheduler tries to assign more tasks to resources with lower quality, thereby decreasing the makespan. The fact that in fig. 8.5b task t12 is assigned to r2 instead of r3 is again due to the probabilistic nature of our method.

8.7.2 Crew Location

In this example we study a scenario where many different tasks must be scheduled in a damage control situation on board of a navy ship. There are routine tasks that are interspersed with damage control tasks. The scenario is based on an example by Kool [2017], extended with several tasks that demonstrate the features of our algorithm. In the scenario five kinds of work must be carried out. A weapon needs to be readied and used, which consists of five tasks. A meeting with the high ranking officers must be scheduled. A cook must prepare food. The deck is divided into 15 sectors that must all be swabbed. In addition to these routine tasks, there has been a grenade impact, and now blanket search must be carried out in order to assess the damage. Blanket search consists of one inspection task per room on the ship.

The weapon tasks demonstrate the resource affinity feature of our algorithm. The scheduler must choose the same weapon for the preparation and usage tasks.

The blanket search tasks demonstrate user-defined quality functions. We assume a simple model of the ship's layout, where 30 rooms are located on one long corridor. All crew members are randomly distributed across the first ten rooms. The quality function evaluates people assigned to blanket search tasks according to the distance between the person and the room, where distance is a less-is-better objective. Calculating the distance between a person and a room in this model is a simple subtraction, but one could imagine a more realistic calculation where distances are calculated using a model of an actual ship with decks, corridors, and staircases.

The cooking, swabbing and meeting tasks are there to provide some noise for the scheduler. The swab deck tasks have no quality function, any crew member is equally suited to do them, even the commander. The staff meeting needs the commander and the section leaders present, there can be no variation in the assignment, the scheduler just has to find a time slot where all required people are free.

Figure 8.6 shows a solution for this scenario. There are some important points to note. The tasks LoadAmmunition, PrepareWeapon, PreActionCalibration, and UseWeapon all need a weapon that can engage small surface targets, and the same weapon should be assigned to all those tasks. After all, it does not make sense to load ammunition for a gun but then engage with another one. In the problem instance this constraint is modelled using resource affinity, and the scheduler only produces solutions where the constraint is satisfied.

The model of the ship has 30 rooms, and the crew is distributed over the first ten. The quality function tries to assign blanket search tasks of rooms to people who are close to the rooms. This is why people in room 10 tend to get more blanket search tasks for rooms 11 to 30 than people in room 1.

8.8 Conclusion and Discussion

We have presented an algorithm that calculates solutions to the C2 scheduling problem. The C2 scheduling problem expresses the scheduling requirements in damage control scenarios on board of navy ships. It is a variant of the MSRCPSP, and extends it with resource affinity constraints and user-defined quality functions. We use an evolutionary algorithm to search the solution space for schedules that have a good score according to the weighted product of all objectives.

We argue that evolutionary algorithms lend itself well for our problem, because we are not only interested in the best schedule, but also in alternative solutions. Evolutionary algorithms calculate sets of solutions, which as a by-product contain the best alternative solutions it has found. Furthermore, evolutionary algorithms do not need knowledge about the function they are trying to optimize. This is useful for our purpose, because user-defined quality functions can be arbitrary Turing-complete calculations, and there can be no general heuristic that could take advantage of them to guide the search.

Coactive design. We envision our algorithm to be part of an integrated mission management system, where the computer is part of the team. Humans and machines should contribute to the joint activity of mission management with their respective strengths. In his Ph.D. thesis, Johnson [2014] argues that for interdependent teamwork, the computer must not be a black box. Its internal process must be exposed. To trust and rely on the computer, it must have three key characteristics: *observability*, *predictability*, and *directability*. Johnson defines them as follows. Observability is making relevant aspects of the machine's status, knowledge and environment available to others. Predictability means that others can rely on their prediction about the machine's behaviour. Directability is the ability to be influenced by others. This can be for example through direct commands, guidance, preferences, or suggestions. We now look at our method with respect to these three characteristics.

Observability in our case means that the user can see what the machine knows about the ship's status, in other words the user should have access to the operational picture. If the user-defined quality functions make use for example of the weather conditions, the location of people on board, or the degradation status of equipment, this information should be accessible in a clear and user-friendly manner. Representation and display of the operational picture is out of the scope of this article, so this point is not applicable to the discussion.

Predictability in our case means that given similar situations, the algorithm should come up with similar solutions. The probabilistic nature of evolutionary algorithms makes our method come off less well in this respect. For some problem instances, where good solutions are isolated between many bad solutions, we observed that it can take a long time or lots of runs to find a good solution again that we have seen before. We believe that this problem can be mitigated by developing parameter sets that are appropriately dimensioned for typical problems.

Directability in our case means that the user can guide the search towards solutions that fulfil certain desired properties. This is where the strength of our method lies. User-defined quality functions, the weighted product, and resource affinity constraints were specifically included to give users instruments to express preferences about the kind of solutions they desire. The example in section 8.7.1 shows how emphasizing the makespan produces quick solutions where less capable resources are being utilized, while de-emphasizing the makespan assigns more work to fewer high-quality resources, resulting in schedules that take longer to complete. The example in section 8.2.3 demonstrates how arbitrary constraints can be encoded in user-defined quality functions, in this case to prevent Bob from having to fly the helicopter.

Planning versus scheduling. In section 8.2.1 we argue that planning and scheduling cannot always be clearly separated. Nonetheless, many such situations can be expressed as C2 scheduling problems, by choosing an appropriate level of abstraction. For example, even though choosing between a helicopter and a boat requires different setup and cleanup

procedures, we can model the situation as three generic tasks: setup, perform, and cleanup. We assume that the personnel is sufficiently educated to know which setup and cleanup procedures are required for either transport. Our resource affinity constraints can be used to guarantee that all three tasks deal with the same transport.

Stopping criterion. In our current implementation, the algorithm stops after a fixed number of generations. This guarantees termination of the algorithm after a pre-determined time, but it might stop at an unfortunate moment where a new area of the search space has been discovered but not yet fully explored. Other stopping criteria have been proposed for evolutionary algorithms. One could use different kinds of convergence criteria, like that the distance between the best and the worst solution in the population falls below a certain threshold, or that the best best solution has not been improved for some time.

Feasibility checking. A problem instance is called *feasible* when it has at least one valid solution. Not every C2 scheduling problem instance is feasible. It could be helpful to check feasibility of a scenario before setting out to finding the best solution. Checking feasibility of C2 problems is tricky, because resources can have multiple capabilities. Take for example a scenario with one task that requires two capabilities A and B, and one resource that has both. Just looking at the cardinalities of the sets of resources with required capabilities is not sufficient. One might conclude that there is one resource with capability A and one resource with capability B, but when assigning the resource to either, it becomes unavailable for the other. Correia et al. [2012] describe a method of checking feasibility for scheduling problems with flexible resources, based on finding a flow in a specifically constructed network.

8.9 Related Work

Our work combines and extends the following existing work. Our chromosome encoding scheme of assignments, and the greedy schedule builder are inspired by the implementation by Myszkowski et al. [2018]. They have a Java implementation of a genetic algorithm that solves the MSRCPSP with fixed costs. They optimize total cost and makespan. In their version of the MSRCPSP, one task has exactly one capability requirement, whereas our tasks can have multiple. One consequence of this is that their chromosomes always encode valid solutions, while our chromosomes might encode invalid solutions, where not all capability requirements have a resource assigned. We therefore had to add a measure of invalidity to our fitness function, that gives a negative score to solutions depending on the number of unassigned capability requirements. Furthermore, our work extends theirs with user-defined quality functions and resource affinity constraints.

The evolutionary algorithm we use is based on the implementation in Haskell by Alexeev [2014]. We chose his implementation over others because we wanted our implementation to be in Clean, and his algorithm was small and simple, which made it easy to port and extend. We extended his work in several ways to make it suitable for our needs. We added support for invalid chromosomes, we preserve the best candidates in each generation, and we support additional parameters to the fitness function besides just the chromosome. Instead of a single number, our fitness function returns a list of objectives, tagged as either less-is-better or more-is-better. Finally, our genetic algorithm calculates the weighted product of the objectives of chromosomes in order to sort them.

Some other work related to ours is as follows. Deb et al. [2000] have a genetic algorithm for solving multi-objective optimization problems. Their algorithm is used for example by Myszkowski et al. [2018]. We chose not to use their algorithm, because it depends on non-dominated sorting, which Deb himself describes as ineffective for more than a handful of objectives. Our problem instances produce dozens of objectives, so we opted for a genetic algorithm that sorts solutions based the weighted product instead.

In his Ph.D. thesis, Casas [2012] focuses on exact methods for solving the MSRCPSP for medium-sized instances. They model the MSRCPSP in five different ways using mixed integer linear programming. They use a technique called *column generation* to estimate lower bounds and solve instances, both exact and heuristically. Finally, they use *recovering beam search* to solve large instances. One of his supervisors wrote her Ph.D. thesis about solving the MSRCPSP [Bellenguez-Morineau and Neron, 2006; Bellenguez-Morineau and Néron, 2007; Bellenguez-Morineau, 2008].

Artigues [2008] has written a book about the resource constrained scheduling problem, which contains a chapter about the multi-skill variant.

Correia et al. [2012] were among the first ones to study solving the multi-skill variant of the RCPSP. In this publication, they call it RCPSP with flexible resources. Almeida et al. [2016] present a parallel scheduling heuristic to solve the MSRCPSP, and conduct computer experiments to validate their approach. Recently they used a genetic algorithm to solve the MSRCPSP [Almeida et al., 2018].

8.10 Future Work

As this chapter focuses on a small problem in the large area of command and control, there are many ways in which our work can be continued.

Re-planning. We envision our algorithm to be used during missions, when some resources are already busy executing tasks. If new tasks must be executed, for example due to an incident that requires damage control, the running tasks should be taken into consideration. Depending on the severity of the incident it might be better to pause some of the running tasks to free their resources, let them work on the incident, and then resume their previous tasks. These task switches could have their own associated costs that the algorithm should try to minimize. One way to implement this could be a measure for the distance between two schedules. If we could calculate some sort of *degree of similarity* between two schedules, it would be possible to let the algorithm compare each candidate with the currently running schedule. Candidates that are more similar to the running schedule should be preferred. As with all other objectives, this similarity measure should be weighted so that users can influence the algorithm to either calculate a faster but disruptive new schedule, or one that does not disrupt the current activities but takes longer, or has lower quality.

Human in the loop. The long-term goal for this work is the development of interactive decision support, which is integrated into a mission management tool. The decision maker should always have the last word, and the tool should allow all decisions to be manually overridden, even if that violates constraints. It is always possible that the constraints are erroneous, for example that a resource can actually be assigned to a task, even if that is not known to the tool. In such a case, humans should not be restricted by the design of the software. This is called *human in the loop*. Future work in this direction should first

identify requirements in which ways humans should be able to change the outcome of the scheduler, and then implement it in a user-friendly and intuitive manner. This could include visualization of the consequences of manual overrides to resources and running tasks.

Influence schedule task ordering. In the current implementation, the evolutionary algorithm calculates an assignment, from which a schedule is built in a greedy manner, where tasks first in the instance definition get scheduled first. This way, task priorities can be modelled by putting tasks with higher priority earlier in the instance definition. The downside is that the scheduler and the quality functions have no influence on the task ordering. It would be nice to be able to let the quality functions evaluate the task ordering. That would make it possible for example to encode dynamic blanket search routes as a scheduling problem. This is a difficult extension to the algorithm however, as the chromosomes need to encode a task ordering that should be guaranteed to respect the ordering imposed by the ordering constraints of the instance.

Other population-based methods. We argued in section 8.5 that evolutionary algorithms are a natural fit for our problem, because they are population-based metaheuristics. We chose an evolutionary algorithm for our proof-of-concept implementation because it is easy to implement and extend. There are other population-based metaheuristics in the literature that could be explored as well, like particle swarm optimization or ant colony optimization. This line of work would require developing a set of example instances that covers many of our intended scenarios and takes our implementation to the computational limit. Then, the other methods have to be implemented as solvers to the C2 scheduling problem. In this way, it would be possible to experimentally compare which metaheuristics is faster and delivers better results.

Acknowledgements

We would like to thank Fok Bolderheij for many hours of fruitful discussion. This research is funded by the Royal Netherlands Navy and TNO.

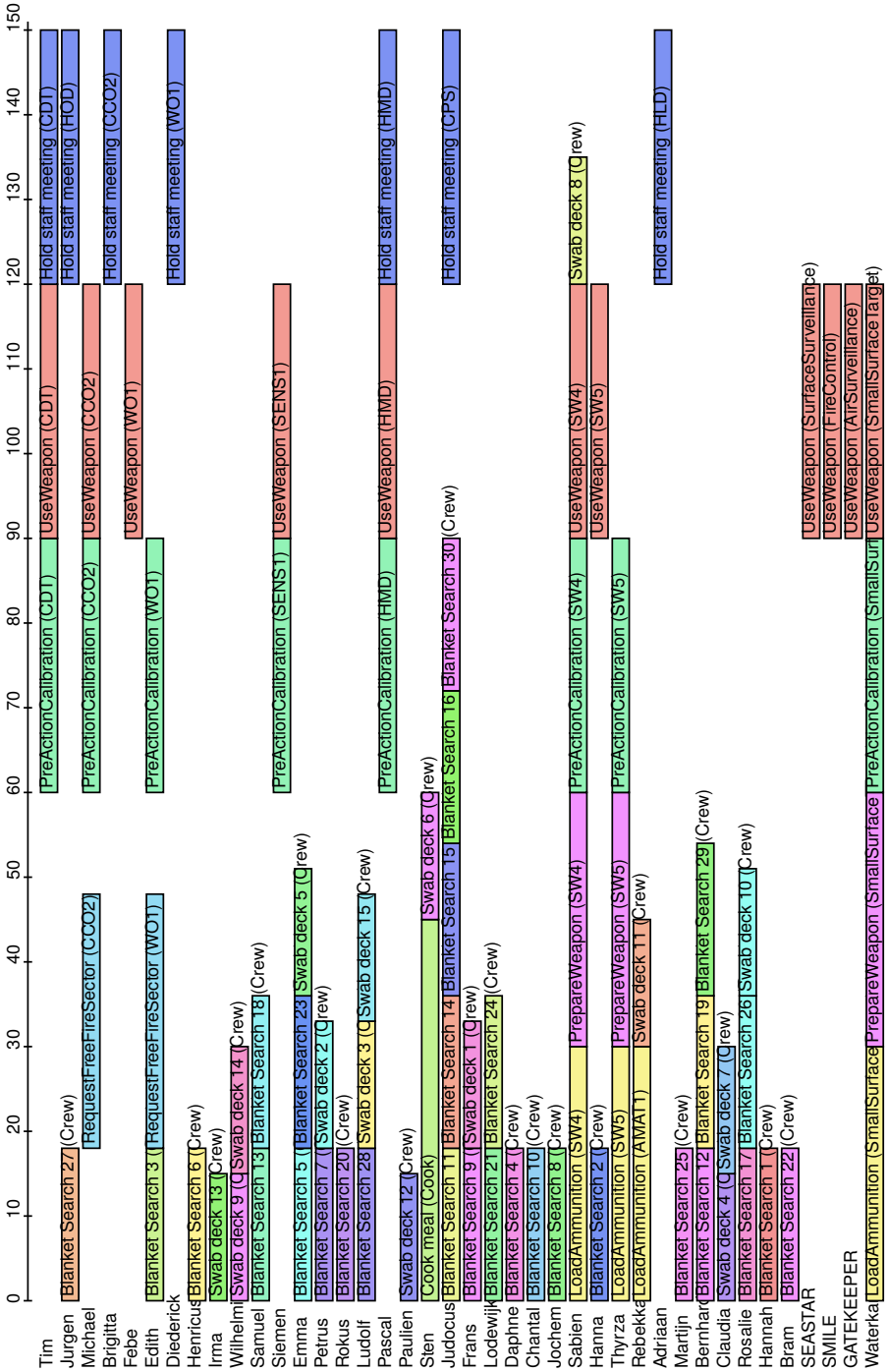


Figure 8.6: A scenario with mixed damage control and routine work. The scenario uses resource affinity and user-defined capability functions.

9 Conclusion

In this thesis I have shown how different techniques from programming language research can be applied to workflows, by means of encoding workflows as task-oriented programs, and then applying the techniques to those programs. I did this to answer three different facets of the central question of whether a plan can reach its goal.

Do we have enough resources to execute a plan?

The static resource analyses, with and without skylines, work nicely in some cases, but result in overapproximation in other cases. This is especially prominent when there is a big difference between the cheapest and most expensive outcome. Polymorphism can make the analysis more precise for let-bound function definitions. When two branches are merged, the result can be a cost that neither of the branches on its own exhibits. And of course, when loops or recursive functions are involved, all bets are off and the analysis has to overapproximate with costs of infinity, unless the analysis can detect that two or more loop iterations do not cost more than the first one.

By using existing programming language techniques, we inherit their strengths but also their weaknesses. The analysis gives upper bounds, which guarantees that if that much resources are available, the workflow can always be executed. How realistic this upper bound is depends on the workflow under analysis, in particular on the cost of rarely used expensive edge cases. I believe that this is still preferable to unsound techniques, like Monte-Carlo simulation, which might miss low-probability edge cases and not report their costs at all.

Drawing skylines over line numbers loses information about the shape of the power consumption over time, but provides insight into which lines of code contribute to the power consumption. For total energy consumption of a program in kilowatt-hours, we must consider the amount of time a certain power draw is active. As we disregard time in the analysis results of SECA, the outcome should be interpreted as momentary power consumption, to detect spikes in unexpected places. Our analysis is no silver bullet that replaces all other analyses, but it gives an additional perspective from which programmers can see their program.

Does a plan reach its goal at all?

My co-authors and me have defined a formal semantics for the programming language TopHat, that captures the essence of task-oriented programming. We found that small-step semantics work nicely for defining semantics driven by user input, and that layered semantics keeps boundaries between language features explicit. iTasks has stable and

unstable values as important concept in the core definition. While designing a minimal language that can still be called task-oriented, we found that this does not need to be in the core definition. Value stability is useful for creating applications, but it can be added in user-space, on top of the core definition. Furthermore, iTasks has two feature-rich basic combinators, parallel and step, which programmers rarely use in their bare form. iTasks provides different programmer-friendly combinators by restricting the basic combinators, and exposing only part of their functionality. This is useful for keeping the implementation of iTasks maintainable and performant, but to see the naked essence of TOP, it is better to have simple combinators with less features, ideally only one, that can be combined using functional programming to form more powerful combinators.

To further pursue the aim of proving properties about workflows, my co-authors and me have extended TopHat with a symbolic execution semantics. This allows programmers to formulate properties about the final task value, which can then be automatically discharged by an SMT solver in conjunction with the path constraint that accompanies every symbolic execution result. If all results satisfy the property, the program itself satisfies the property. With symbolic execution comes the problem of state space explosion, but it is convenient because programmers do not have to explicitly state invariants, and do not have to manually construct proofs. While symbolic execution is a computationally expensive method, we believe that it is handy for programmers to use.

Can a plan be repaired?

I performed requirements engineering, working together with people from the Netherlands Defence Academy, to identify how task-oriented programming can be used in the construction of an integrated mission management system. We found that the most helpful component of such a system would be decision support in hectic situations, when suddenly additional tasks must be executed while some resources become unavailable. A scheduler that assigns resources to tasks while respecting arbitrary constraints and optimizing user-defined quality metrics would be of great usefulness. Because of the open-endedness of these requirements, we realized that a heuristic scheduler that does not directly inspect the constraints and quality metrics would be our best bet. I implemented a genetic algorithm that can be parametrized by such constraints and metrics. It calculates a whole set of possible schedules. While there are no guarantees that the method finds the best solution, or any solution at all, it is an easy to implement way to navigate large search spaces. The algorithm finds good schedules for our example problems in a matter of seconds.

Bibliography

Wil M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.

Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

Peter Achten, Marko van Eekelen, and Rinus Plasmeijer. Generic graphical user interfaces. In *Proc. 15th IFL*, pages 152–167. Springer, 2003.

Peter Achten, Marko van Eekelen, Rinus Plasmeijer, and Arjen van Weelden. Arrows for generic graphical editor components. Technical Report NIII-R0416, University of Nijmegen, 2004.

Susanne Albers. Energy-efficient algorithms. *Communications of the ACM*, 53(5):86–96, 2010.

Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In *Formal Methods for Components and Objects*, volume 5382, pages 113–132. Springer, 2008.

Alexander Alexeev. Simple parallel genetic algorithm implementation. `https://github.com/afiskon/simple-genetic-algorithm`, 2014. Accessed: 2019-05-20.

Artem Alimarine and Rinus Plasmeijer. A generic programming extension for clean. In *Proc. 13th IFL*, pages 168–185. Springer, 2001.

Bernardo F. Almeida, Isabel Correia, and Francisco Saldanha da Gama. Priority-based heuristics for the multi-skill resource constrained project scheduling problem. *Expert Syst. Appl*, 57:91–103, 2016.

Bernardo F. Almeida, Isabel Correia, and Francisco Saldanha da Gama. A biased random-key genetic algorithm for the project scheduling problem with flexible resources. *TOP: An Official Journal of the Spanish Society of Statistics and Operations Research*, 26(2): 283–308, July 2018. doi: 10.1007/s11750-018-0472-9.

T. Amtoft, F. Nielson, H. R. Nielson, and J. Ammann. Polymorphic subtyping for effect analysis: The dynamic semantics. *LNCS*, 1192, 1997.

Heinrich Apfelmus. reactive-banana project homepage. `https://wiki.haskell.org/Reactive-banana`, 2019. Accessed 2019-02-13.

Arduino AG. Arduino project hub. `https://create.arduino.cc/projecthub`. Accessed: 2020-05-01.

Christian Artigues. *The Resource-Constrained Project Scheduling Problem*. HAL CCSD; ISTE-WILEY, 2008.

David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. *Theoretical Computer Science.*, 389(3): 411–445, December 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2007.09.003.

Robert Atkey. Amortised resource analysis with separation logic. In *Programming Languages and Systems*, volume 6012 of *LNCS*, pages 85–103. Springer, 2010. ISBN 978-3-642-11956-9. doi: 10.1007/978-3-642-11957-6.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

Odile Bellenguez-Morineau. Methods to solve multi-skill project scheduling problem. *4OR*, 6(1):85–88, 2008.

Odile Bellenguez-Morineau and Emmanuel Neron. Genetic algorithms for the multi-skill project scheduling problem. *Tenth International Workshop on Project Management and Scheduling, Poznan, Poland*, pages 73–77, April 2006.

Odile Bellenguez-Morineau and Emmanuel Néron. A branch-and-bound method for solving multi-skill project scheduling problem. *RAIRO - Operations Research*, 41(2):155–170, 2007.

Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

Gérard Berry and Manuel Serrano. Hop and HipHop: Multitier web orchestration. *CoRR*, abs/1312.0078, 2013.

Gérard Berry, Cyprien Nicolas, and Manuel Serrano. HipHop: A synchronous reactive extension for Hop. In *Proceedings of the 1st International Workshop on Programming Language and Systems Technologies for Internet Clients*, pages 49–56. ACM, 2011.

F. Bolderheij, J.M. Jansen, A.A. Kool, and J. Stutterheim. A mission-driven C2 framework for enabling heterogeneous collaboration. In *Netherlands Annual Review of Military Studies*, NL ARMS, The Hague, 2018. Asser Press. doi: 10.1007/978-94-6265-246-0_6.

Fok Bolderheij. *Mission-Driven Sensor Management, Analysis, Design, Implementation and Simulation*. PhD thesis, Delft University of Technology, 2007.

Frédéric Boussinot and Robert De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

John Boyd. Destruction and creation. U.S. Army Command and General Staff College, 1976.

Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT - a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM. doi: 10.1145/800027.808445.

Jeffrey M. Bradshaw, Robert R. Hoffman, David D. Woods, and Matthew Johnson. The seven deadly myths of "autonomous systems". *IEEE Intelligent Systems*, 28(3):54–61, 2013.

David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *SIGARCH Computer Architecture News*, 28(2):83–94, May 2000.

Peter Brucker, Bernd Jurisch, and Andreas Krämer. Complexity of scheduling problems with multi-purpose machines. *Annals of Operations Research*, 70(0):57–73, 1997. ISSN 1572-9338.

Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *Architectural Support for Programming Languages and Operating Systems*, pages 239–254, 2014. doi: 10.1145/2541940.2541977.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. 8th OSDI'08*, pages 209–224, San Diego, California, USA, December 2008. USENIX Association.

Carlos Eduardo Montoya Casas. *New methods for the multi-skills project scheduling problem. (Nouvelles méthodes pour le problème de gestion de projet multi-compétence)*. PhD thesis, École des mines de Nantes, France, 2012.

Stephen Chang, Alex Knauth, and Emina Torlak. Symbolic types for lenient symbolic execution. *PACMPL*, 2(POPL):40:1–40:29, 2018.

Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. *SIGPLAN Notices*, 47(10):831–850, October 2012. ISSN 0362-1340.

Gregory Cooper and Shriram Krishnamurthi. Frtime: Functional reactive programming in plt scheme. Technical Report CS-03-20, Department of Computer Science, Brown University, Rhode Island, 2004.

Isabel Correia, Lídia Lampreia Lourenço, and Francisco Saldanha da Gama. Project scheduling with flexible resources: formulation and inequalities. *OR Spectrum*, 34(3): 635–663, 2012.

Patrick Cousot and Radhia Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. In *Logics and Languages for Reliability and Security*, volume 25, pages 1–29. IOS Press, 2010.

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *Proc. ESOP 2005*, volume 3444, pages 21–30. Springer, 2005.

Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, Chichester, UK, 2001.

Kalyanmoy Deb. Multi-objective optimization using evulutionary algorithms: An introduction. Technical report, Indian Institute of Technology Kanpur, 2011.

Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II. In *Proceedings of PPSN VI*, volume 1917 of *LNCS*, pages 849–858, Paris, France, September 2000. Springer-Verlag. ISBN 3-540-41056-2.

Thierry Despeyroux. Executable specification of static semantics. In *Proc. of Int. Symp. Semantics of Data Types*, volume 173 of *LNCS*, pages 215–233. Springer, 1984.

Allen B. Downey. *The Little Book of Semaphores*. Green Tea Press, 2008.

Christoph Dürr, Sigrid Knust, Damien Prot, and Óscar C. Vásquez. The scheduling zoo. `http://schedulingzoo.lip6.fr/`, 2016. Accessed: 2019-01-31.

Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of ICFP'97, Amsterdam, The Netherlands*, pages 263–273, 1997.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of ICFP'02*, pages 48–59, 2002.

Bernard van Gastel. *Assessing sustainability of software - Analysing Correctness, Memory and Energy Consumption*. PhD thesis, Open University, 2016.

Bernard van Gastel, Rody Kersten, and Marko van Eekelen. Using dependent types to define energy augmented semantics of programs. In *Proceedings of FOPARA'15*, volume 9964 of *LNCS*, pages 20–39, 2015. ISBN 978-3-319-46558-6.

Tobias Gedell, Jörgen Gustavsson, and Josef Svenningsson. Polymorphism, subtyping, whole program analysis and accurate data types in usage analysis. In *APLAS 2006, Sydney, Australia*, pages 200–216. Springer, 2006.

Ilche Georgievski and Marco Aiello. HTN planning: Overview, comparison, and beyond. *Artif. Intell*, 222:124–156, 2015.

Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

Aggelos Giantsios, Nikolaos Papaspyrou, and Konstantinos Sagonas. Concolic testing for functional languages. *Sci. Comput. Program.*, 147:109–134, 2017. doi: 10.1016/j.scico.2017.04.008.

Haack and Wells. Type error slicing in implicitly typed higher-order languages. *SCIPROG: Science of Computer Programming*, 50, 2004.

Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. A generic usage analysis with subeffect qualifiers. In *Proceedings of ICFP'07, Freiburg, Germany*, pages 235–246. ACM, 2007.

William T. Hallahan, Anton Xue, and Ruzica Piskac. Building a symbolic execution engine for Haskell. In *Proceedings of TAPAS 17*, 2017.

William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. Lazy counterfactual symbolic execution. In *Proceedings of PLDI 2019*, page 411–424, New York, NY, USA, 2019. ACM. doi: 10.1145/3314221.3314618.

John Harrison. Formal verification. In *Software and Systems Safety - Specification and Verification*, volume 30, pages 103–157. IOS Press, 2011.

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall Int'l, 1985.

Tony Hoare. An axiomatic basis for computer programming. *CACM: Communications of the ACM*, 12, 1969.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of POPL'11*, pages 357–370. ACM, 2011. ISBN 978-1-4503-0490-0.

Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. TRACER: A symbolic execution tool for verification. In *Computer Aided Verification (23ed CAV'12)*, volume 7358 of *LNCS*, pages 758–766. Springer, July 2012.

Mauro Jaskelioff, Neil Ghani, and Graham Hutton. Modularity and implementation of mathematical operational semantics. *Electr. Notes Theor. Comput. Sci.*, 229(5):75–95, 2011.

Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. Estimating the worst-case energy consumption of embedded software. In *Proceedings of RTAS'06*, pages 81–90. IEEE, 2006. doi: 10.1109/RTAS.2006.17.

Matthew Johnson. *Coactive Design: Designing Support for Interdependence in Human-Robot Teamwork*. PhD thesis, Technische Universiteit Delft, 2014.

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. *ACM SIGPLAN Notices*, 45(1):223–236, January 2010.

Meuse N. O. Junior, Silvino Neto, Paulo Romero Martins Maciel, Ricardo Massa Ferreira Lima, Angelo Ribeiro, Raimundo S. Barreto, Eduardo Tavares, and Frederico Braga. Analyzing software performance and energy consumption of embedded systems by probabilistic modeling: An approach based on coloured Petri nets. In *Proceedings of ICATPN'06*, pages 261–281, 2006.

Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Trade-Offs*. Cambridge University Press, 1993.

Steven Kerrison, Umer Liqat, Kyriakos Georgiou, Alejandro Serrano Mena, Neville Grech, Pedro Lopez-Garcia, Kerstin Eder, and Manuel V. Hermenegildo. Energy consumption analysis of programs based on XMOS ISA-level models. In *Proceedings of LOPSTR'13*. Springer, September 2013.

Rody Kersten, Olha Shkaravska, Bernard van Gastel, Manuel Montenegro, and Marko van Eekelen. Making resource analysis practical for real-time Java. In *Proceedings of JTRES'12*, pages 135–144. ACM, 2012. doi: 10.1145/2388936.2388959.

Rody Kersten, Paolo Parisen Toldin, Bernard van Gastel, and Marko van Eekelen. A Hoare logic for energy consumption analysis. In *Proceedings of FOPARA'13*, volume 8552 of *LNCS*, pages 93–109. Springer, 2014.

James C. King. A new approach to program testing. *SIGPLAN Not.*, 10(6):228–233, April 1975. ISSN 0362-1340. doi: 10.1145/390016.808444.

James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19 (7):385–394, July 1976.

Markus Klinik. Resource analysis source code repository. `https://gitlab.science.ru.nl/mklinik/program-analysis`, 2017. Accessed: 2020-07-24.

Markus Klinik. GA-scheduler source code repository. `https://gitlab.science.ru.nl/mklinik/ga-scheduler`, 2019. Accessed: 2020-07-24.

Markus Klinik, Bernard van Gastel, Cynthia Kop, and Marko van Eekelen. SECA project website. `https://gitlab.science.ru.nl/mklinik/eca-symbolic-execution/-/wikis/home`, a. Accessed: 2020-02-06.

Markus Klinik, Bernard van Gastel, Cynthia Kop, and Marko van Eekelen. SECA source code repository. `https://gitlab.science.ru.nl/mklinik/eca-symbolic-execution`, b. Accessed: 2020-01-29.

Markus Klinik, Jurriaan Hage, Jan Martin Jansen, and Rinus Plasmeijer. Predicting resource consumption of higher-order workflows. In *Proceedings of PEPM 2017, Paris, France*, pages 99–110. ACM, 2017a. ISBN 978-1-4503-4721-1.

Markus Klinik, Jan Martin Jansen, and Rinus Plasmeijer. The sky is the limit: Analysing resource consumption over time using skylines. In *Proceedings of the 29th IFL*. ACM, 2017b.

Markus Klinik, Jan Martin Jansen, and Fok Bolderheij. Dynamic resource and task management. In *Netherlands Annual Review of Military Studies*, NL ARMS, pages 91–105, The Hague, 2018. Asser Press.

Markus Klinik, Jan Martin Jansen, and Rinus Plasmeijer. Resource scheduling with computable constraints for maritime command and control. In *MAST Asia 2019*, Makuhari Messe, Chiba, Tokyo, Japan, 2019.

Markus Klinik, Bernard van Gastel, Cynthia Kop, and Marko van Eekelen. Skylines for symbolic energy consumption analysis. In *Proceedings of the 25th FMICS*. Springer, 2020.

Bram Kool. Integrated mission management voor C2-ondersteuning. Bachelor's thesis, Netherlands Defence Academy, Den Helder, The Netherlands, 2017.

Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. A task-based DSL for microcomputers. In *Proc. of the Real World Domain Specific Languages Workshop, RWDSL@CGO 2018, Vienna, Austria*, pages 4:1–4:11, 2018. doi: 10.1145/3183895.3183902.

Pieter W. M. Koopman, Artem Alimarine, Jan Tretmans, and Marinus J. Plasmeijer. Gast: Generic automated software testing. In *Proceedings of IFL'02*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2002.

Pieter W. M. Koopman, Rinus Plasmeijer, and Peter Achten. An executable and testable semantics for iTasks. In *Proceedings of the 20th IFL*, 2008.

Ruud Koot and Jurriaan Hage. Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. In *Proceedings of PEPM'15, Mumbai, India*, pages 127–138. ACM, 2015.

Robbert Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University, Nijmegen, The Netherlands, 2015.

Marek Kubale. The complexity of scheduling independent two-processor tasks on dedicated processors. *Information Processing Letters*, 24(3):141–147, 1987.

Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.

Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

Bas Lijnse and Rinus Plasmeijer. iTasks 2: iTasks for end-users. In *Revised Selected Papers of the 21st IFL*, volume 6041 of *LNCS*, pages 36–54. Springer, 2009.

Bas Lijnse, Jan Martin Jansen, and Rinus Plasmeijer. Incidone: A task-oriented incident coordination tool. In *Proceedings of ISCRAM*, 2012.

Simon Marlow et al. *Haskell 2010 language report*, 2010. `http://www.haskell.org/`.

Raúl Mencía, María R. Sierra, Carlos Mencía, and Ramiro Varela. Schedule generation schemes and genetic algorithm for the scheduling problem with skilled operators and arbitrary precedence relations. In *Proc. of ICAPS*, pages 165–173. AAAI Press, 2015.

Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992. doi: 10.1109/2.161279.

Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *Proc. 24th OOPSLA, Orlando, Florida, USA*, pages 1–20, 2009.

ATmega48A/PA/88A/PA/168A/PA/328/P Data Sheet. Microchip Technology Inc., 2018.

Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

Peter D. Mosses. Formal semantics of programming languages: An overview. *Electron. Notes Theor. Comput. Sci.*, 148(1):41–73, 2006.

Christian J. Muise, J. Christopher Beck, and Sheila A. McIlraith. Optimal partial-order plan relaxation via maxsat. *J. Artif. Intell. Res*, 57:113–149, 2016.

Pawel B. Myszkowski, Maciej Laszczyk, Ivan Nikulin, and Marek Skowronski. iMOPSE: a library for bicriteria optimization in multi-skill resource-constrained project scheduling problem. *Soft Computing*, 2018.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare type theory. *ACM SIGPLAN Notices*, 41(9):62–73, September 2006. doi: 10.1145/1159803.1159812.

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *Proceedings of the 13th ICFP*, pages 229–240, 2008. doi: 10.1145/1411204.1411237.

Nico Naus. *Assisting End Users in Workflow Systems*. PhD thesis, Utrecht University, Utrecht, The Netherlands, 2020.

Nico Naus and Johan Jeuring. Building a generic feedback system for rule-based problems. In *Proc. 17th TFP, College Park, MD, USA*, volume 10447 of *LNCS*, pages 172–191. Springer, 2016. ISBN 978-3-030-14804-1.

Nico Naus, Tim Steenvoorden, and Markus Klinik. A symbolic execution semantics for TopHat. In *Proceedings of the 31st IFL*, School of Computing, National University of Singapore, 2019.

Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Higher order symbolic execution for contract verification and refutation. *J. Funct. Program*, 27, 2017.

Flemming Nielson, Hanne Riis Nielson, and Torben Amtoft. Polymorphic subtyping for effect analysis: The algorithm. In *Proc. 5th LOMAPS*, volume 1192 of *LNCS*, pages 207–243. Springer, 1996a.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

Hanne Riis Nielson and Flemming Nielson. *Semantics With Applications: A Formal Introduction*. Wiley, 1992.

Hanne Riis Nielson, Flemming Nielson, and Torben Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In *Proc. 5th LOMAPS*, volume 1192 of *LNCS*, pages 141–171. Springer, 1996b.

Bruno Nogueira, Paulo Maciel, Eduardo Tavares, Ermeson Andrade, Ricardo Massa, Gustavo Callou, and Rodolfo Ferraz. A formal model for performance and energy evaluation of embedded systems. *EURASIP Journal on Embedded Systems*, pages 2:1–2:12, January 2011. ISSN 1687-3955. doi: 10.1155/2011/316510.

OASIS. Web services business process execution language. `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`, 2019. Accessed: 2019-02-12.

Object Management Group. Business process model and notation (BPMN) version 1.2. Technical report, Object Management Group, 2009.

OPLSS. Programming Languages Background 1 - Robert Harper - OPLSS 2017. `https://www.youtube.com/watch?v=pzIkOhf3gX0`, 2017. Accessed: 2020-06-19.

Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proc. PLDI'15*, pages 346–356. ACM, June 2015.

Simon Peyton-Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering Theories of Software Construction*, NATO ASI Series, pages 47–96. IOS Press, 2001. Marktoberdorf Summer School 2000.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

Rinus Plasmeijer and Peter Achten. The implementation of iData - a case study in generic programming. In *Proc. 17th IFL*. University of Dublin, 2005.

Rinus Plasmeijer and Marko van Eekelen. Clean language report (version 2.1). `http://clean.cs.ru.nl`, 2002.

Rinus Plasmeijer, Peter Achten, and Javier Pomer Tendillo. iData for the world wide web - generic programming techniques for high-level server-side web scripting. Technical report, University of Nijmegen, 2005.

Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: Executable specifications of interactive work flow systems for the web. In *Proc. ICFP'07*, pages 141–152. ACM, 2007.

Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, Thomas van Noort, and John van Groningen. iTasks for a change: Type-safe run-time change in dynamically evolving workflows. In *Proc. of PEPM '11, Austin, TX, USA*, pages 151–160. ACM Press, 2011.

Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter W. M. Koopman. Task-oriented programming in a pure functional language. In *Proc. of PPDP'12, Leuven, Belgium*. ACM, 2012.

Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

Gordon D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.

Radboud University. Institute for computing and information science (iCIS) research data management policy. `https://www.ru.nl/icis/research-data-management/`, 2020. Accessed: 2020-07-21.

Parthasarathy Ranganathan. Recipe for efficiency: principles of power-aware computing. *Communications of the ACM*, 53(4):60–67, 2010.

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: approximate data types for safe and general low-power computation. *SIGPLAN Notices*, 46(6):164–174, June 2011. ISSN 0362-1340.

Eric Saxe. Power-efficient software. *Communications of the ACM*, 53(2):44–48, 2010. ISSN 0001-0782. doi: 10.1145/1646353.1646370.

Dana S. Scott. Domains for denotational semantics. In *Proc. of 9th Colloquium on Automata, Languages and Programming, Aarhus, Denmark*, volume 140 of *LNCS*, pages 577–613. Springer, 1982.

Dana S. Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, volume 21. Polytechnic Institute of Brooklyn, 1971.

Amit Sinha and Anantha P. Chandrakasan. JouleTrack: A web based tool for software energy profiling. In *Proceedings of DAC'01*, pages 220–225. ACM, 2001.

Tim Steenvoorden and Nico Naus. TopHat source code repository. `https://github.com/timjs/tophat-haskell`, 2019. Accessed: 2020-07-24.

Tim Steenvoorden, Nico Naus, and Markus Klinik. TopHat: A formal foundation for task-oriented programming. In *Proceedings of the 21st PPDP*, Porto, Portugal, 2019.

S. S. Stevens. Mathematics, measurement and psychophysics. *Handbook of experimental psychology*, pages 1–49, 1951.

Jurriën Stutterheim. *A Cocktail of Tools*. PhD thesis, Radboud University, Nijmegen, The Netherlands, 2017.

Jurriën Stutterheim, Rinus Plasmeijer, and Peter Achten. Tonic: An infrastructure to graphically represent the definition and behaviour of tasks. In *Proc. of 15th TFP, Soesterberg, The Netherlands*, pages 122–141, 2014.

Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. Maintaining separation of concerns through task oriented software development. In *Proc. of 18th TFP, Canterbury, UK*, 2017.

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *Proc. PLDI '13*, pages 387–398, 2013. doi: 10.1145/2491956.2491978.

Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

Wouter Swierstra. A hoare logic for the state monad. In *Proc. TPHOLs*, volume 5674 of *LNCS*, pages 440–451. Springer, 2009. ISBN 978-3-642-03358-2.

Chris Tofallis. Add or multiply? A tutorial on ranking and choosing with multiple criteria. *INFORMS Trans. Education*, 14(3):109–119, 2014.

Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Proc. of 15th IFL*, volume 3145 of *LNCS*, pages 86–101. Springer, 2003.

Patrick Weaver. A brief history of scheduling. In *myPrimavera Conference*, 2006.

Jeroen Weijers, Jurriaan Hage, and Stefan Holdermans. Security type error diagnosis for higher-order, polymorphic languages. *Sci. Comput. Program*, 95:200–218, 2014.

C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012.

Acknowledgements

Als erstes möchte ich mich bei meinen Eltern Veronika und Josef bedanken, die mich allzeit ohne Wenn und Aber auf meinem Weg von Erlangen nach Nürnberg, Bochum, Tennenlohe, Nijmegen, und schließlich Eindhoven unterstützt haben.

First of all I would like to thank my parents Veronika and Josef, who always unconditionally supported me on my journey from Erlangen to Nürnberg, Bochum, Tennenlohe, Nijmegen, and finally Eindhoven.

Many thanks also to Terry Stroup for more than one and a half decades of friendship, fuelled by and fuelling my interest in theory of programming languages and compiler construction, mathematics, and Italian eating habits.

Thanks to my supervisor Rinus, who offered me the PhD position, supported me along the way, and had my back in several tough disputes with a certain critical sponsor.

I would also like to thank my girlfriend Priyanka for her continued support and warmth and love.

I will always remember hiking, dining, and playing with the MAMS club. Thank you Manxia, Anna, and Sven for many wonderful weekends together.

Trying to tackle a PhD on one's own is a drab endeavour. Thanks to Paul Fiterau-Brostean for making me shift gears, and thanks to my co-authors Jurriaan Hage, Tim Steenvoorden, Nico Naus, Bernard van Gastel, Cynthia Kop, Fok Bolderheij, Jan-Martin Jansen, and Marko van Eekelen for making the endeavour not only bearable, but actually enjoyable.

Many thanks to Pieter Koopman, who employed me in his project when my official contract ended, which gave me the time to finish my PhD.

Finally, in no particular order, thanks to all the people I met in and around Nijmegen, who were good colleagues, or good friends, or both. Sjaak Smetsers, Peter Achten, Ingrid Berenbroek, Jurriën Stutterheim, Bas Lijnse, Mart Lubbers, Laszlo Domoszlai, Giso and Amy Dal, Chris Elings, Joshua Moerman and Tessa Matser, Kelley van Evert, Jurriaan Rot, Ralf Hinze, Henning Basold, Marcos Bueno, Sven-Bodo Scholz, John van Groningen, Niels van der Weide, Herman Geuvers, Steffen Michels, Pol van Aubel, Nils Jansen, Gabriel Bucur, Jacopo Acquarelli, Michael Colesky, Alexis Linard, Dennis Groß, Hans-Nikolai Viessmann.

Research Data Management

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of Radboud University, The Netherlands [Radboud University, 2020].

The following research datasets have been produced during this PhD research:

- Chapter 2: Klinik, M.A.A.; Hage, J.; Jansen, J.M.; Plasmeijer, M.J. (2020). Source code of the resource analysis compiler. DANS EASY. 10.17026/dans-zdg-y4sg.
- Chapter 3 Klinik, M.A.A.; Jansen, J.M.; Plasmeijer, M.J. (2020). Source code of the resource-time analysis compiler. DANS EASY. 10.17026/dans-xvb-b9w9.
- Chapter 4: Klinik, M.A.A.; Gastel, B.E. van; Kop, C.L.M. (2020). Source code for the SECA symbolic execution engine. DANS EASY. 10.17026/dans-29u-kyh8.
- Chapter 5: Steenvoorden, T.J.; Naus, N.; Klinik, M.A.A. (2020). Source code for the TopHat implementation. DANS EASY. 10.17026/dans-zks-p82q.
- Chapter 6: Naus, N.; Steenvoorden, T.J.; Klinik, M.A.A. (2020). Source code for the symbolic TopHat execution engine. DANS EASY. 10.17026/dans-zub-xac3.
- Chapter 8: Klinik, M.A.A.; Jansen, J.M.; Plasmeijer, M.J. (2020). Source code for the genetic algorithm scheduler. DANS EASY. 10.17026/dans-xya-4urm.

Curriculum Vitae

Markus Klinik

1981	Born on April 16, Nürnberg, Germany
2000 - 2006	Georg Simon Ohm Fachhochschule Nürnberg, computer science Graduation as Diplom-Informatiker FH
2006 - 2007	Areva, Erlangen, Germany Software Developer signal processing, embedded systems
2007 - 2012	Elektrobit, Tennenlohe, Germany Software Developer automotive embedded systems, infotainment
2012 - 2014	Radboud University, Nijmegen, The Netherlands MSc Mathematical Foundations of Computer Science (MFoCS)
2014 - 2020	Radboud University, Nijmegen, The Netherlands PhD Candidate
since 2021	Chordify, Utrecht, The Netherlands Software Developer web backend